清華大學 计算机系列教材

陈渝 向勇 编著

操作系统实验指导



清华大学出版社

清华大学计算机系列教材

操作系统实验指导

陈　渝　向　勇　编著

清 华 大 学 出 版 社
北　　京

内 容 简 介

本书是操作系统课程的实验教材，旨在帮助读者加强对操作系统原理与设计实现的理解，以分析、设计、改进和实现一个微型但全面的操作系统——ucore 为基本目标，通过增量式地完成 8 个基于 ucore 操作系统实验为操作系统实践环节，最终让读者了解并掌握操作系统的原理、设计与实现。

本书强调对于操作系统动手实践，是对操作系统实践教学的一次探索，可作为高等院校计算机专业操作系统课程的实验教材，也可作为各类操作系统教学的培训教材及自学参考资料。

图书在版编目（CIP）数据

操作系统实验指导 / 陈渝，向勇编著．—北京：清华大学出版社，2013.7
清华大学计算机系列教材
ISBN 978-7-302-32777-6

Ⅰ．①操…　Ⅱ．①陈…　②向…　Ⅲ．①操作系统－高等学校－教学参考资料　Ⅳ．①TP316

中国版本图书馆 CIP 数据核字（2013）第 136214 号

责任编辑：白立军
封面设计：常雪影
责任校对：梁　毅
责任印制：刘海龙

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：北京国马印刷厂
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：12.75　　**字　　数**：319 千字
版　　次：2013 年 7 月第 1 版　　**印　　次**：2013 年 7 月第 1 次印刷
印　　数：1～2000
定　　价：25.00 元

产品编号：053017-01

序

“清华大学计算机系列教材”已经出版发行了30余种，包括计算机科学与技术专业的基础数学、专业技术基础和专业等课程的教材，覆盖了计算机科学与技术专业本科生和研究生的主要教学内容。这是一批至今发行数量很大并赢得广大读者赞誉的书籍，是近年来出版的大学计算机专业教材中影响比较大的一批精品。

本系列教材的作者都是我熟悉的教授与同事，他们长期在第一线担任相关课程的教学工作，是一批很受本科生和研究生欢迎的任课教师。编写高质量的计算机专业本科生（和研究生）教材，不仅需要作者具备丰富的教学经验和科研实践，还需要对相关领域科技发展前沿的正确把握和了解。正因为本系列教材的作者们具备了这些条件，才有了这批高质量优秀教材的产生。可以说，教材是他们长期辛勤工作的结晶。本系列教材出版发行以来，从其发行的数量、读者的反映、已经获得的国家级与省部级的奖励，以及在各个高等院校教学中所发挥的作用上，都可以看出本系列教材所产生的社会影响与效益。

计算机学科发展异常迅速，内容更新很快。作为教材，一方面要反映本领域基础性、普遍性的知识，保持内容的相对稳定性；另一方面，又需要紧跟科技的发展，及时地调整和更新内容。本系列教材都能按照自身的需要及时地做到这一点。如王爱英教授等编著的《计算机组成与结构》、戴梅萼教授等编著的《微型计算机技术及应用》都已经出版了第四版，严蔚敏教授的《数据结构》也出版了三版，使教材既保持了稳定性，又达到了先进性的要求。

本系列教材内容丰富，体系结构严谨，概念清晰，易学易懂，符合学生的认知规律，适合教学与自学，深受广大读者的欢迎。系列教材中多数配有丰富的习题集、习题解答、上机及实验指导和电子教案，便于学生理论联系实际地学习相关课程。

随着我国进一步的开放，我们需要扩大国际交流，加强学习国外的先进经验。在大学教材建设上，我们也应该注意学习和引进国外的先进教材。但是，“清华大学计算机系列教材”的出版发行实践以及它所取得的效果告诉我们，在当前形势下，编写符合国情的具有自主版权的高质量教材仍具有重大意义和价值。它与国外原版教材不仅不矛盾，而且是相辅相成的。本系列教材的出版还表明，针对某一学科培养的要求，在教育部等上级部门的指导下，有计划地组织任课教师编写系列教材，还能促进对该学科科学、合理的教学体系和内容的研究。

我希望今后有更多、更好的我国优秀教材出版。

清华大学计算机系教授，中国科学院院士

张钹

前　　言

对于在校的学生和工程技术人员而言，能否有效地了解操作系统原理后面的具体设计实现呢？陆游说过："纸上得来终觉浅，绝知此事要躬行"。我们在教学过程中，也深刻体会到这一点。我们认为，在了解基本的操作系统概念和原理的基础上，通过实际动手来一步一步分析、设计和实现一个微型化的操作系统，会深入了解操作系统的实现细节，并体会到概念原理和实际实现之间的紧密联系及巨大差异。

操作系统是一个复杂系统软件，涉及内容繁多，发展也很快，如 Linux、Windows 等，都是上百万行的源代码规模。开发人员开发这些操作系统软件的目的是用于实际计算机系统中，而不是用于教学，所以直接用这些操作系统来分析了解操作系统的实现和进行操作系统实验会比较复杂。而且目前部分操作系统教材的内容也越来越庞大和抽象，而面向操作系统设计实现的实验部分相对就少了很多。这两方面交织在一起，导致学生了解和掌握操作系统的实际细节很困难。

早期的 UNIX 操作系统实现和 MIT 教授 Frans Kaashoek 等基于 UNIX v6 设计的 xv6 操作系统给了我们启发：对一个计算机专业的本科生而言，在了解操作系统原理的基础上，设计实现一个操作系统有挑战，但是可行！我们对此进行了尝试与探索，以设计实现一个微型但全面的操作系统——ucore 为基本目标，以增量式递进开发方式完成各种基于 ucore 操作系统的实验为实践过程，以在此过程中逐步介绍的操作系统的基本概念和原理为实践指导，做到有"理"可循和有"码"可查，最终让读者了解和掌握操作系统的原理、设计与实现。目前的实验内容包含如下 8 个。

（1）启动操作系统的 bootloader：了解操作系统启动前的状态和要做的准备工作。

（2）物理内存管理子系统：理解硬件段/页模式和操作系统如何管理物理内存。

（3）虚拟内存管理子系统：理解页表机制、缺页故障处理以及内存替换算法。

（4）内核线程子系统：理解相对简单的内核态线程的动态管理过程。

（5）用户进程管理子系统：理解用户态进程动态管理过程以及系统调用过程。

（6）处理器调度子系统：理解操作系统的调度过程和调度算法。

（7）同步互斥与进程间通信子系统：理解进程间如何同步互斥以及进行信息交换和共享。

（8）文件系统：理解文件系统的具体实现，与进程管理和内存管理等的关系。

其中每个开发步骤都是建立在上一个步骤之上的，就像搭积木，从一个一个小木块，最终搭出来一个小房子。在搭房子的过程中，完成从理解操作系统原理到实践操作系统设计与实现的探索过程。最新的代码和文档放在 http://www.github.com/chyyuu/ucore_lab 上。如果有同学和 OS 爱好者觉得这些实验难度不够，大家可参加更有挑战和乐趣的 ucore plus 实验，这些实验位于 http://www.github.com/chyyuu/ucore_plus 下。目前的代码和文档还有许多不完善和错误的地方需要改进，欢迎大家批评指正。

在实现基于 ucore 的操作系统实验过程中，我们参考和借鉴了 xv6、OS161 以及 Linux 的设计思路和实现代码，而且 Frans Kaashoek 博士也亲自给予了帮助与指导。国内多所高校的老师，包括陈向群、王雷、陈鹏、陈莉君、原仓周、蒲晓蓉等都给予了指导和帮助。操作系统课程的助教王乃峥、袁昕颢、茅俊杰、陈宇恒、曹聪、杨杨等完成了大量工作，在此表示衷心的感谢！

陈　渝　向　勇

2013 年 3 月 12 日

目　录

第 1 章　实验 0：操作系统实验准备

1.1　实 验 目 的

（1）了解操作系统开发的实验环境。
（2）熟悉命令行方式的编译、调试工程。
（3）掌握基于硬件模拟器的调试技术。
（4）熟悉 C 语言编程和指针的概念。
（5）了解 x86 汇编语言。

1.2　准 备 知 识

1.2.1　了解 OS 实验

编写一个操作系统程序难吗？别被现在上百万行的 Linux 和 Windows 操作系统吓倒。当年 Thompson 在他夫人带着小孩度假留他一人在家时，编写了 UNIX；当年 Linus 还是一个 21 岁大学生时弄出了 Linux 的雏形。站在这些巨人的肩膀上，我们能否也尝试一下做“巨人”的滋味呢？

MIT 的 Frans Kaashoed 教授等在 2006 年左右参考 PDP-11 上的 UNIX Version 6 写了一个可在 x86 上跑的 xv6 操作系统（基于 MIT License），用于学生学习操作系统。Harvard 大学的 David A. Holland 等也设计了 OS161 操作系统用于操作系统实验教学。我们可以站在他们的肩膀上，参考他们的设计思路、方法和源代码，尝试着一步一步完成一个从“空空如也”到“五脏俱全”的“麻雀”操作系统——ucore，此“麻雀”OS 包含软件启动、中断处理、物理内存管理、虚存管理、进程管理、处理器调度、同步互斥、进程间通信、文件系统等主要操作系统内核功能，每个实验包含的内核代码量（C＋asm＋注释）在 300～10 000 行左右，充分体现了“小而全”的指导思想。

ucore 的运行环境可以是真实的 x86 计算机，不过考虑到调试和开发的方便，我们可采用 x86 硬件模拟器，比如 QEMU、BOCHS、VirtualBox、VMware Player 等。ucore 的开发环境主要是 GCC 中的 gcc、gas、ld 和 MAKE 等工具，也可采用集成了这些工具的 IDE 开发环境 Eclipse-CDT 等。在分析源代码上，可以采用 Scitools 提供的 understand 软件（跨平台），Windows 环境上的 Source Insight 软件，或者基于 emacs＋ctags、vim＋ctags 等，都可以比较方便地在一堆文件中查找变量、函数定义、调用/访问关系等。软件开发的版本管理可以采用 GIT、SVN 等。比较文件和目录的不同可发现不同实验中的差异性和进行文件合并操作，可使用 meld、kdiff3、UltraCompare 等软件。调试（Deubg）实验有助于发现设计中的错误，可采用 gdb（配合 qemu）等调试工具软件。整个实验的运行环境和开发环境既可以

在 Linux 使用,又可以在 Windows 中使用。推荐使用 Linux 环境。

如何一步一步来实现 ucore 呢? 根据一个操作系统的设计实现过程,可以有如下的实验步骤。

(1) 启动操作系统的 bootloader,用于了解操作系统启动前的状态和要做的准备工作,了解运行操作系统的硬件支持,操作系统如何加载到内存中,理解外设中断和陷阱中断等。

(2) 物理内存管理子系统,用于理解 x86 分段/分页模式,了解操作系统如何管理物理内存。

(3) 虚拟内存管理子系统,通过页表机制和换入换出(Swap)机制,以及故障中断和缺页故障处理等,实现基于页的内存替换算法。

(4) 内核线程子系统,用于了解如何创建相对与用户进程更加简单的内核态线程,如对内核线程进行动态管理等。

(5) 用户进程管理子系统,用于了解用户态进程创建、执行、切换和结束的动态管理过程,了解在用户态通过系统调用得到内核态的内核服务的过程。

(6) 处理器调度子系统,用于理解操作系统的调度过程和调度算法。

(7) 同步互斥与进程间通信子系统,了解进程间如何进行信息交换和共享,并了解同步互斥的具体实现以及对系统性能的影响,研究死锁产生的原因,以及如何避免死锁。

(8) 文件系统,了解文件系统的具体实现,与进程管理等的关系,了解缓存对操作系统 I/O 访问的性能改进,了解虚拟文件系统(VFS)、Buffer Cache 和 Disk Driver 之间的关系。

其中每个开发步骤都是建立在上一个步骤之上,就像搭积木,从一个一个小木块,最终搭出来一个小房子。在搭房子的过程中,完成从理解操作系统原理到实践操作系统设计与实现的探索过程。这个房子最终的建筑架构和建设进度如图 1-1 所示。

如果完成上述实验后还想完成更大的挑战实验,那么可以参加 ucore 的研发项目,我们可以完成 ucore 的网络协议栈,增加图形系统,增加编程语言支持(比如目前的 golang、python 等),在 ARM 嵌入式系统上运行,支持虚拟机功能等。这些项目已经有同学参与,欢迎其他有兴趣的同学加入!

接下来将介绍实验环境的设置、Linux 系统的安装、Linux 命令行的使用方法、各种实验工具的使用方法以及 Intel 80386 硬件的重要特征等。这些内容足以写成另外几本书,这里主要是介绍与实验相关的内容并进行了大量的精简,部分内容来源于 Internet(如 Ubuntu forum 网站、qemu 网站、GNU 网站等)和 Intel 的 CPU 手册,由于内容繁多,无法给出具体的参考署名,这里对相关作者一并表示感谢。

1.2.2 设置实验环境

我们参考了 MIT 的 xv6、Harvard 的 OS161 和 Linux 等设计了 ucore OS 实验,所有 OS 实验需在 Linux 下运行。对于经验不足的同学,推荐参考“通过虚拟机使用 Linux 实验环境”一节用虚拟机方式进行实验。

1. 开发 OS 实验的简单步骤

在 github 网站(https://github.com/chyyuu/ucore_pub)可下载我们提供的 lab1～

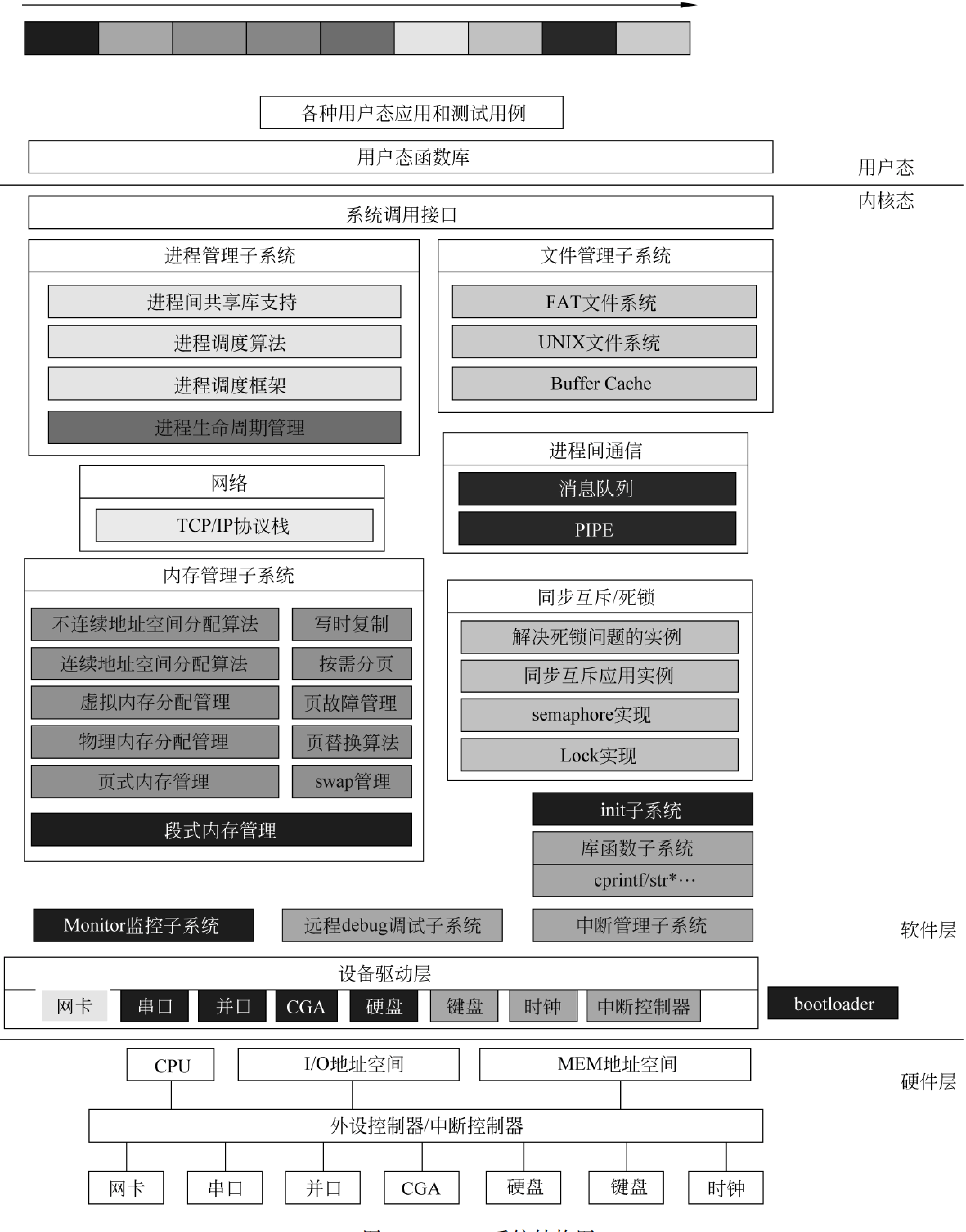


图 1-1　ucore 系统结构图

lab8 实验相关软件和文档，大致经过以下过程就可以完成使用。

（1）下载并解压软件包。

（2）进入各个 OS 实验工程目录，如执行“cd Code//Lab1”可查看目录下 Lab1 的源码，

执行"cd doc"查看各个实验相关文档。

(3) 根据实验要求阅读源码并修改代码(用各种代码分析工具和文本编辑器)。

(4) 并编译源码,例如执行:make。

(5) 如编译不过则返回步骤(3)。

(6) 如编译通过则可查看运行情况,如执行:make qemu。

(7) 如想测试个人的实验完成是否基本正确,则可执行:make grade。

(8) 如果实现基本正确可查看运行情况(即看到步骤(6)的输出存在不是 OK 的情况)则返回步骤(3)。

(9) 如果实现基本正确(即看到步骤(6)的输出都是 OK)则生成实验提交软件包,例如,执行:make handin。

(10) 把生成的使用提交软件包和实验报告上传或发电子邮件给助教和老师。

另外,可以通过 make debug 或 make debugnox 命令实现通过 gdb 远程调试 OS 实验工程。

2. 通过虚拟机使用 Linux 实验环境(推荐:最容易的实验环境安装方法)

这是最简单的一种通过虚拟机方式使用 Linux 并完成 OS 各个实验的方法,Linux 操作系统和各种实验所需的开发软件都已经安装到虚拟硬盘文件中了,只需安装 vir tual Box 虚拟机软件就可以开始进行实验了。配置的大体步骤如下。

(1) 安装 VirtualBox 虚拟机软件(有 Windows 版本和其他 OS 版本,可到 http://www.virtualbox.org/wiki/Downloads 下载)。

(2) 在网上下载一个已经安装好各种所需编辑/开发/调试/运行软件的 Linux 实验环境的 VirtualBox 虚拟硬盘文件(oslabs_for_student_2012.zip,包含一个虚拟磁盘镜像文件和两个配置描述文件,下载此文件的网址参见 https://github.com/chyyuu/ucore_lab 下的 README 中的描述)。

(3) 用 2345 好压软件(有 Windows 版本,可到 http://www.haozip.com 下载。一般软件解压不了 xz 格式的压缩文件,也可以用其他支持解压 zip 和 xz 压缩格式的软件)先解压到 C 盘(也可以是其他盘符路径)的 vms 目录下,即:

```
C:\vms\ubuntu-12.04.vbox.xz
C:\vms\ubuntu-12.04.vmdk.vmdk.xz
C:\vms\ubuntu-12.04.vmdk- flat.vmdk.xz
```

在分别用好压软件或其他能够解压 xz 压缩格式的软件进一步解压上述三个文件,形成

```
C:\vms\ubuntu-12.04.vbox
C:\vms\ubuntu-12.04.vmdk.vmdk
C:\vms\ubuntu-12.04.vmdk-flat.vmdk
```

解压后这三个文件所占用的硬盘空间为 12GB 左右。在 VirtualBox 中加载 ubuntu-12.04.vbox,就可以启动并运行 Linux 实验环境了。

启动到提示输入用户名时,请输入:

```
chy
```

当提示输入口令时，只需按空格键和 Enter 键即可。然后就进入开发环境中了。实验内容位于 ucore_lab 目录下。可以通过如下命令获得放在 github 上的整个实验的最新代码和文档：

```
$ git clone https://github.com/chyyuu/ucore_lab.git
```

并可通过如下命令获得以后更新后的代码和文档：

```
$ git pull
```

如需要进一步了解一下 git 的基本使用方法，这可以通过网络搜索获得很多这方面的信息。

3. 安装 Linux 实验环境(适合自己安装 Linux 系统的同学)

这里主要以 Ubuntu Linux 12.04(32 位)作为整个实验的系统软件环境。首先需要安装 Ubuntu Linux 12.04，这里主要介绍一种比较容易的 WUBI Linux 的安装方式。

WUBI 方式安装(最容易的 Linux 安装方法)

WUBI 是一个专门针对 Windows 用户的 UBUNTU Linux 安装工具，读者需要做的只是单击几下鼠标而已。不需要改变分区设置，不需要启动文件，不需要 Live CD。使用 WUBI 可很方便地安装或卸载 Ubuntu，如果读者从来没有安装过 UBUNTU Linux，WUBI 很适合初学者第一次安装 UBUNTU Linux。具体方法如下。

(1) 去 OS Course FTP 或官方网站 http://releases.ubuntu.com/12.04/ubuntu-12.04-desktop-i386.iso 下载一个 ubuntu-12.04-desktop-i386 的 ISO 文件。

(2) 通过 winrar 等工具将下载来的 ISO 文件中的 wubi.exe 解压出来，放在任意一个分区的根目录下。这里推荐预留了一个至少大小为 8GB 的 NTFS 分区，单击 wubi.exe 安装文件，这时会弹出对话框。

注意：*在 ubuntu 12.04 中，“在 Windows 内安装”的那个选项被禁用了，只能通过以下指令开启(假定 X 为光驱盘符)：X:\wubi.exe-force-wubi。*

(3) 设置好分区将要安装的分区、语言、分配的系统大小、用户名和密码(务必记住)之后，单击“安装”按钮，这时如果正在安装的计算机已经联网了，会自动从镜像网站上下载 ISO 文件。这里采用绕过 WUBI 下载镜像 ISO 的方法安装 Ubuntu 12.04，会节省大量时间。避免下载 ISO 文件的这一步非常关键。在进行这一步之前请将网线断开，然后将提前下载来的 ubuntu-12.04-desktop-i386.iso 文件复制到 WUBI 所创建的 Ubuntu 目录下的 install 文件夹中，重新运行 wubi.exe。这次再也不会提示下载 ISO 文件了。几秒钟后，WUBI 就会提示你重新启动系统。注意，此时 Ubuntu 并没有安装在硬盘上，必须重新启动才开始进行 Ubuntu 12.04 的安装。

(4) 单击“完成”按钮，选择重启计算机。计算机重启后，在启动选项中选择 Ubuntu，出现“press 'Esc' to…” 时，不用理会，这时 Ubuntu 滚动条出现在屏幕上。此时，才正式开始安装 Ubuntu 12.04 至硬盘分区某一目录下。接下来什么也不用做，只需等待。当提示正式安装完成后，重新启动计算机系统，可以发现在启动选项中有“Ubuntu”和“Windows”。可以根据读者的情况进行选择。

4. 使用 Linux

在实验过程中，需要了解基于命令行方式的编译、调试、运行操作系统的实验方法。为此，需要了解基本的 Linux 命令行使用。

（1）命令模式的基本结构和概念。Ubuntu是当前易于使用和操作的Linux发行版。Linux的命令的操作模式功能可以实现各种功能。简单地说,命令行就是基于字符命令的用户界面,也被称为文本操作模式。绝大多数情况下，用户通过输入一行或多行命令直接与计算机互动。

（2）如何进入命令模式。假设Ubuntu Linux启动后进入图形界面,单击左上角,在提示行输入并回车,从而可以在此软件界面中进行命令行操作。

打开gnome-terminal程序后可能会注意到类似下面的界面：

```
chy@ chyhome-PC:~ $ ls
file1.txt file2.txt file3.txt tools
```

“chy@ chyhome-PC:～\$”这些字符串被称为命令终端提示符,它表示计算机已经就绪,正在等待着用户输入操作指令。以作者的屏幕画面为例,chy是当前所登录的用户名,chyhome-PC是这台计算机的主机名,～表示当前目录。此时输入任何指令按Enter键之后该指令将会提交到计算机运行,比如可以输入命令：ls再按下Enter键：

```
ls [Enter]
```

注意：[Enter]是指输入完ls后按下Enter键,而不是输入这个单词,ls这个命令将会列出当前所在目录里的所有文件和子目录列表。

下面介绍Bash Shell程序的基本使用方法,它是Ubuntu默认的外壳程序。

1）常用指令

（1）查询文件列表：ls。

```
chy@ chyhome-PC:~ $ ls
file1.txt file2.txt file3.txt tools
```

ls命令默认状态下将按首字母升序列出当前文件夹下面的所有内容,但这样直接运行所得到的信息是比较少的,通常它可以结合以下这些参数运行以查询更多的信息。

① ls /：将列出根目录“/”下的文件清单。如果给定一个参数,则命令行会把该参数当作命令行的工作目录。换句话说,命令行不再以当前目录为工作目录。

② ls -l：将列出一个更详细的文件清单。

③ ls -a：将列出包括隐藏文件(以“.”开头的文件)在内的所有文件。

④ ls -lh：将以KB/MB/GB的形式给出文件大小,而不是以纯粹的Bytes。

（2）查询当前所在目录：pwd。

```
chy@ chyhome-PC:~ $ pwd
/home/chy
```

（3）进入其他目录：cd。

```
chy@ chyhome-PC:~ $ pwd
/home/chy
chy@ chyhome-PC:~ $ cd /root/
chy@ chyhome-PC:~ $ pwd
/root
```

上面的例子中，当前目录原来是/home/chy，执行 cd /root/之后再运行 pwd 可以发现，当前目录已经改为/root 了。

（4）在屏幕上输出字符：echo。

```
chy@chyhome-PC:~$echo"Hello World"
Hello World
```

这是一个很有用的命令，它可以在屏幕上输入指定的字符串（""中的内容），可用于信息提示和输出相关内容等要求。

（5）显示文件内容：cat。

```
chy@chyhome-PC:~$cat file1.txt
Roses are red.
Violets are blue,
and you have the bird-flue!
```

也可以使用 less 或 more 命令来显示比较大的文本文件内容。

（6）复制文件：cp。

```
chy@chyhome-PC:~$cp file1.txt file1_copy.txt
chy@chyhome-PC:~$cat file1_copy.txt
Roses are red.
Violets are blue,
and you have the bird-flue!
```

这个命令可以复制一个文件的内容到另一个文件中。

（7）移动文件：mv。

```
chy@chyhome-PC:~$ls
file1.txt
file2.txt
chy@chyhome-PC:~$mv file1.txt new_file.txt
chy@chyhome-PC:~$ls
file2.txt
new_file.txt
```

这个命令可以简单理解为一个文件改名字。

注意：在命令行模式进行操作时，系统基本上不会给丰富的提示信息，当然，绝大多数的命令可以通过加上一个参数-v 来要求系统给出执行命令的反馈信息。

```
chy@chyhome-PC:~$mv-v file1.txt new_file.txt
'file1.txt'->'new_file.txt'
```

（8）建立一个内容为空的文本文件：touch。

```
chy@chyhome-PC:~$ls
file1.txt
chy@chyhome-PC:~$touch tempfile.txt
chy@chyhome-PC:~$ls
```

```
file1.txt
tempfile.txt
```

如果查看 tempfile. txt 的内容,可以发现此文件没有存放任何信息。

(9) 建立一个目录: mkdir。

```
chy@chyhome-PC:~$ls
file1.txt
tempfile.txt
chy@chyhome-PC:~$mkdir test_dir
chy@chyhome-PC:~$ls
file1.txt
tempfile.txt
test_dir
```

这个命令可用来创建文件或目录。

(10) 删除文件/目录: rm。

```
chy@chyhome-PC:~$ls  -p
file1.txt
tempfile.txt
test_dir/
```

ls 命令的—p 参数可以给目录显示增加一个"/",这样可以更清楚看出一个文件是否是目录文件。

```
chy@chyhome-PC:~$rm-i tempfile.txt
rm: remove regular empty file 'test.txt'?y
chy@chyhome-PC:~$ls-p
file1.txt
test_dir/
```

可以看到 test. txt 文件已经被删除了。

```
chy@chyhome-PC:~$rm test_dir
rm: cannot remove 'test_dir': Is a directory
chy@chyhome-PC:~$rm-R test_dir
```

删除一个目录需要增加新的参数"-R"或"-r",这样才能删除此目录下的所有文件和目录本身。使用此命令需要小心。

```
chy@chyhome-PC:~$ls-p
file1.txt
```

可以看到 test dir 目录文件已经被删除了。

在上面的操作中,首先通过 ls 命令查询可知当前目下有两个文件和一个文件夹。

① 可以用参数"-p"来让系统显示某一项的类型,比如是文件/文件夹/快捷链接等。

② 用 rm -i 尝试删除文件,"-i"参数是让系统在执行删除操作前输出一条确认提示;i(Interactive)也就是交互性的意思。

③ 当人们尝试用上面的命令去删除一个文件夹时会得到错误的提示，因为删除文件夹必须使用“-R”或“-r”(Recursive，循环)参数。

特别提示：在使用命令行操作方式时，系统假设操作人员很明确自己在做什么，它不会给太多的提示，比如执行 rm-Rf/，它将会删除硬盘上所有的东西，并且不会给出任何提示，所以，尽量在使用命令时加上“-i”的参数，以让系统在执行前进行一次确认，防止执行误操作。如果觉得每次都要输入“-i”太麻烦，可以执行以下的设置别名命令 alias，让“-i”成为 rm 命令的默认参数：

```
alias rm='rm-i'
```

(11) 查询当前进程：ps。

```
chy@chyhome-PC:~$ps
PID TTY          TIME CMD
21071 pts/1    00:00:00 bash
22378 pts/1    00:00:00 ps
```

这条命令会列出当前运行中的所有进程。

注意：这个命令的参数没有“-”。

① “ps-a”可以列出系统当前运行的所有进程，包括由其他用户启动的进程。

② “ps auxww”会列出除一些很特殊进程以外的所有进程，并会以一个可读的形式显示结果，每一个进程都会有较为详细的解释。

基本命令的介绍就到此为止，可以访问网络得到更加详细的 Linux 命令介绍。

2) 控制流程

(1) 输入输出。input 用来读取通过键盘(或其他标准输入设备)输入的信息，output 用于在屏幕(或其他标准输出设备)上输出指定的输出内容。另外还有一些标准的出错提示也是通过这个命令来实现的。通常在遇到操作错误时，系统会自动调用这个命令来输出标准错误提示。

能重定向命令中产生的输入和输出流的位置。

(2) 重定向。如果想把命令产生的输出流导入一个文件而不是(默认的)终端，可以使用如下的语句：

```
chy@chyhome-PC:~$ls>file4.txt
chy@chyhome-PC:~$cat file4.txt
file1.txt  file2.txt file3.txt
```

如果 file4.txt 不存在的话，以上例子将创建文件 file4.txt。

注意：如果 file4.txt 已经存在，那么上面的命令将覆盖文件的内容。如果想将内容添加到已存在的文件内容的末尾，可以用下面的命令模式：

```
command>>filename
```

示例：

```
chy@chyhome-PC:~$ls>>file4.txt
chy@chyhome-PC:~$cat file4.txt
```

```
file1.txt   file2.txt file3.txt
file1.txt   file2.txt file3.txt file4.txt
```

在这个例子中，会给原有的文件中添加了新的内容。接下来会见到另一种重定向方式：将把一个文件的内容作为将要执行的命令的输入。以下是这个命令模式：

```
command< filename
```

示例：

```
chy@ chyhome- PC:~ $ cat> file5.txt
a3.txt
a2.txt
file2.txt
file1.txt
<Ctrl- D> //这表示敲入 Ctrl+ D键
```

在上述命令中，通过 cat 和>命令创建了一个文件 file5. txt。

```
chy@ chyhome- PC:~ $ sort< file5.txt
a2.txt
a3.txt
file1.txt
file2.txt
```

在上述命令中，sort 接受 file5. txt 的每一行的内容，并按照字母排序，最后再输出到屏幕上。

(3) 管道。Linux 的强大之处在于它能把几个简单的命令联合起来成为功能强大的命令，这通过键盘上的管道符号"|"完成。现在，来排序上面的 grep 命令：

```
chy@ chyhome- PC: ~ $ grep txt< file5.txt| sort> file6.txt
```

在上述命令组合中，grep 命令搜索 file5 中包含"txt"的字符串，将找到的包含"txt"的输出行通过"|"管道命令传递给 sort 命令，sort 命令对这些字符串行进行排序，并把排好序的字符串行写入到文件 file6. txt。

另外一个小例子：有时候用 ls 命令列出的文件信息很多，超过了一屏，导致看起来很不方便，这时"|"管道命令也可以发挥作用，尝试执行"ls-1|less"可一页一页地看信息。

(4) 后台命令。通过后台进程的方式，可以在命令行并发运行多个命令。方法很简单，要启动一个命令到后台执行，只需在通常的命令后追加一个 & 即可。

```
sleep 60 &
ls
```

睡眠命令 sleep 在后台运行，操作人员依然可以与计算机交互。除了不同步启动命令以外，最好把 & 理解成";"。

如果有一个前台命令将占用很多时间，您想把它放入后台运行。只要在命令运行时按下 Ctrl+Z 的组合键，它就会挂起暂停。然后输入 bg 命令，可把当前挂起的这个前台命令转入后台执行。如果想让它再转到前台执行，可通过执行 fg 命令使其转回前台。

```
sleep 60
<ctrl-z>            //这表示敲入 Ctrl+Z组合键
bg
fg
```

另外，如果不想让一个前台命令继续执行，可以使用 Ctrl＋C 组合键来杀死当前的前台命令。

3）环境变量

特殊变量。PATH、PS1、…

4）不显示中文

可通过执行如下命令避免显示乱码中文。在一个 shell 中，执行：

```
export LANG=""
```

这样在这个 Shell 中，output 信息默认是英文。

5）获得软件包

（1）命令行获取软件包。

Ubuntu 下可以使用 apt-get 命令，apt-get 是一条 Linux 命令行命令，适用于 deb 包管理式的操作系统，主要用于自动从互联网软件库中搜索、安装、升级以及卸载软件或者操作系统。一般需要 root 执行权限，所以一般跟随 sudo 命令，例如：

```
sudo apt-get install gcc\Enter\
```

常见的以及常用的 apt 命令如下。

① apt-get install＜package＞：下载＜package＞以及所依赖的软件包，同时进行软件包的安装或者升级。

② apt-get remove＜package＞：移除＜package＞以及所有依赖的软件包。

③ apt-cache search＜pattern＞：搜索满足＜pattern＞的软件包。

④ apt-cache show/showpkg＜package＞：显示软件包＜package＞的完整描述。

例如：

```
chy@chyhome-PC:~$apt-cache show gcc
gcc-4.6-The GNU C compiler
gcc-4.6-base-The GNU Compiler Collection (base package)
gcc-4.6-doc-Documentation for the GNU compilers (gcc, gobjc, g++)
gcc-4.6-multilib-The GNU C compiler (multilib files)
gcc-4.6-source-Source of the GNU Compiler Collection
gcc-4.6-locales-The GNU C compiler (native language support files)
chy@chyhome-PC:~$
```

（2）图形界面软件包获取。

新立得软件包管理器是 Ubuntu 下面管理软件包的图形界面程序，相当于命令行中的 apt 命令。进入方法可以是

```
菜单栏>系统管理>新立得软件包管理器(System>Administration>Synaptic Package Manager)
```

使用更新管理器可以通过标记选择适当的软件包进行更新操作。

6）配置升级源

Ubuntu 的软件包获取依赖升级源，可以通过修改 /etc/apt/sources.list 文件来修改升级源（需要 root 权限）；或者修改新立得软件包管理器中"设置"→"软件库"。

7）查找帮助文件

Ubuntu 下提供 man 命令以完成帮助手册得查询。man 是 manual 的缩写，通过 man 命令可以对 Linux 下常用命令、安装软件，以及 C 语言常用函数等进行查询，获得相关帮助。例如：

```
cchy@chyhome-PC:~$man printf
PRINTF(1)BSD General Commands Manual PRINTF(1)

NAME
⋮
```

通常可能会用到的帮助文件，例如：

```
gcc-doc cpp-doc glibc-doc
```

上述帮助文件可以通过 apt-get 命令或者软件包管理器获得。获得以后可以通过 man 命令进行命令或者参数查询。

5. 实验中可能使用的软件

1）编辑器

（1）Ubuntu 中自带的编辑器可以作为代码编辑的工具。例如，gedit 是 Gnome 桌面环境下兼容 UTF-8 的文本编辑器。它十分简单易用，有良好的语法高亮显示，对中文支持很好。通常可以通过双击或者命令行打开目标文件进行编辑。

（2）Vim 编辑器：Vim 是一款方便的文本编辑软件，是 UNIX 下的同类型软件 VI 的改进版本。Vim 经常被看成"专门为程序员打造的文本编辑器"，功能强大且方便使用，便于进行程序开发。

Ubuntu 下默认安装的 VI 版本较低，功能较弱，建议在系统内安装或者升级到最新版本的 Vim。

① 关于 Vim 的常用命令以及使用，可以通过网络进行查找。

② 配置文件：Vim 的使用需要配置文件进行设置，例如：

```
set nocompatible
set encoding=utf-8
set fileencodings=utf-8,chinese
set tabstop=4
set cindent shiftwidth=4
set backspace=indent,eol,start
autocmd Filetype c set omnifunc=ccomplete#Complete
autocmd Filetype cpp set omnifunc=cppcomplete#Complete
set incsearch
set number
```

```
set display=lastline
set ignorecase
syntax on
set nobackup
set ruler
set showcmd
set smartindent
set hlsearch
set cmdheight=1
set laststatus=2
set shortmess=atI
set formatoptions=tcrqn
set autoindent
```

可以将上述配置文件保存到：

```
~/.vimrc
```

注意：.vimrc 默认情况下隐藏不可见，可以在命令行中通过“ls-a”命令进行查看。如果～目录下不存在该文件，可以手动创建。修改该文件以后，重启 Vim 可以使配置生效。

2）exuberant-ctags

exuberant-ctags 可以为程序语言对象生成索引，其结果能够被一个文本编辑器或者其他工具简捷迅速地定位。支持的编辑器有 Vim、Emacs 等。

实验中，在需要建立索引的 lab 源码目录下，可以使用：

```
chy@chyhome-Pc~/ucore/ab/code/|ab1$ ctags-R
```

来对工程文件建立索引。

默认的生成文件为 tags（可以通过-f 来指定 tags 的索引文件名），在相同路径下执行 vim 命令可以让 Vim 软件使用该索引文件，在 Vim 的编辑模式下，可方便地定位源码中的声明和定义等，例如：

使用 Ctrl＋]组合键可以跳转到相应的声明或者定义处，使用 Ctrl＋t 组合键返回（查询堆栈）等。

提示：习惯 GUI 方式的同学，可采用图形界面的 Understand、Source Insight 等软件。

3）diff&patch

diff 为 Linux 命令，用于比较文本或者文件夹差异，可以通过 man 来查询其功能以及参数的使用。使用 patch 命令可以对文件或者文件夹应用修改。

例如，实验 2 中会继承应用前一个实验 lproja 中对某些文件进行的修改，可以使用如下命令：

```
diff-ruP lab1_origlab1_new>diff.patch
cd lab2
patch-p1-u<../diff.patch
```

注意：lab1_orig 指 lab1 的源文件目录，即未经修改的源码包，lab1_new 是修改后的 lab1 的源文件目录。第一条命令是递归的比较文件夹差异，并将结果重定向输出到

diff.patch 文件中;第三条命令是将 lab1 的修改应用到 lab2 文件夹中的代码中。

提示:习惯 GUI 方式的读者,可采用图形界面的 Meld、Kdiff3、UltraCompare 等软件。

1.2.3 了解编程开发调试的基本工具

在 Ubuntu Linux 中的 C 语言编程主要基于 GNU C 的语法,通过 gcc 来编译并生成最终执行文件。GNU 汇编(Assembler)采用的是 AT&T 汇编格式,Microsoft 汇编采用 Intel 格式。

1. gcc 的基本用法

如果还没装 gcc 编译环境或不确定是否安装,可先执行:

```
chy@chyhome-PC: ~$ sudo apt-get install build-essential
```

来确保安装了编译所需的 gcc 等各种软件工具。注意:sudo 命令的含义是已 root 特权用户的身份执行 apt-get 命令。如果系统提示要输入口令,则只需输入一个空格即可。

1) 编译简单的 C 程序

C 语言经典的入门例子是 Hello world,下面是一示例代码:

```
#include<stdio.h>
int
main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

假定该代码存为文件 hello.c,要用 gcc 编译该文件,使用下面的命令:

```
chy@chyhome-PC: ~ $gcc-Wall hello.c-o hello
```

该命令将文件 hello.c 中的代码编译为机器码并存储在可执行文件 hello 中。机器码的文件名是通过**-o** 选项指定的。该选项通常作为命令行中的最后一个参数。如果被省略,输出文件默认为 a.out。

注意,如果当前目录中与可执行文件重名的文件已经存在,它将被覆盖。

选项"**-Wall**"开启编译器几乎所有常用的警告——**建议你始终使用该选项**。编译器有很多其他的警告选项,但"**-Wall**"是最常用的。默认情况下 gcc 不会产生任何警告信息。当编写 C 或 C++ 程序时编译器警告非常有助于检测程序存在的问题。

本例中,编译器使用了"**-Wall**"选项而没产生任何警告,因为示例程序是完全合法的。

要运行该程序,输入可执行文件的路径如下:

```
chy@chyhome- PC: ~$ ./hello
Hello, world!
```

这将可执行文件载入内存,并使 CPU 开始执行其包含的指令。路径./指代当前目录,因此**./hello** 载入并执行当前目录下的可执行文件 hello。

2）AT&T 汇编基本语法

Ucore 中用到的是 AT&T 格式的汇编，与 Intel 格式的汇编有一些不同。二者在语法上主要有以下几个不同。

（1）寄存器命名原则。

```
AT&T: %eax                    Intel: eax
```

（2）源/目的操作数顺序。

```
AT&T: movl %eax, %ebx         Intel: mov ebx, eax
```

（3）常数/立即数的格式。

```
AT&T: movl$_value, %ebx       Intel: mov eax, _value
```

（4）把 value 的地址放入 eax 寄存器。

```
AT&T: movl$0xd00d, %ebx       Intel: mov ebx, 0xd00d
```

（5）操作数长度标识。

```
AT&T: movw %ax, %bx           Intel: mov bx, ax
```

（6）寻址方式。

```
AT&T: immed32(basepointer, indexpointer, indexscale)
Intel: [basepointer+indexpointer×indexscale+imm32)
```

如果操作系统工作于保护模式下，用的是 32 位线性地址，所以在计算地址时不用考虑 segment:offset 的问题。上式中的地址应为

```
imm32+basepointer+indexpointer×indexscale
```

① 直接寻址。

AT&T：_boo ；_boo 是一个全局的 C 变量。注意加上 $ 是表示地址引用，不加是表示值引用。对于局部变量，可以通过堆栈指针引用。

```
Intel: [_boo]
```

② 寄存器间接寻址。

```
AT&T: (%eax)
Intel: [eax]
```

③ 变址寻址。

```
AT&T: _variable(%eax)
Intel: [eax+_variable]
AT&T: _array(,%eax,4)
Intel: [eax×4+_array]
AT&T: _array(%ebx,%eax,8)
Intel: [ebx+eax×8+_array]
```

3）GCC 内联汇编

基本的 GCC 内联汇编很简单，一般是按照下面的格式：

```
asm("statements");
```

例如：

```
asm("nop"); asm("cli");
```

asm 和__asm__的含义是完全一样的。如果有多行汇编，则每一行都要加上“\n\t”。其中的“\n”是换行符，“\t”是 tab 符。在每条命令的结束加这两个符号，是为了让 gcc 把内联汇编代码翻译成一般的汇编代码时，能够保证换行和留有一定的空格。例如：

```
asm( "pushl %eax\n\t"
    "movl$ 0,%eax\n\t"
    "popl %eax"
    );
```

实际上 gcc 在处理汇编时，是要把 asm(…)的内容“打印”到汇编文件中，所以格式控制字符是必要的。再例如：

```
asm("movl %eax, %ebx");
asm("xorl %ebx, %edx");
asm("movl$ 0, _boo);
```

在上面的例子中，在内联汇编中改变了 edx 和 ebx 的值，但是由于 gcc 的特殊的处理方法(即先形成汇编文件，再交给 gas 汇编软件去分析此汇编文件)，所以 gas 汇编软件并不知道我们已经改变了 edx 和 ebx 的值，如果程序的上下文需要 edx 或 ebx 作暂存，这样就会引起严重的后果，对于变量 _boo 也存在同样的问题。为了解决这个问题，就要用到扩展 GCC 内联汇编语法来避免潜在的访问冲突问题。

4）扩展 GCC 内联汇编

使用扩展 GCC 内联汇编的例子如下：

```
#define read_cr0() ({ \
    unsigned int __dummy; \
    __asm__( \
        "movl %%cr0,%0\n\t" \
        :"=r" (__dummy));\
    __dummy; \
})
```

这里需要从其基本格式来分析其含义。扩展 GCC 内联汇编的基本格式如下：

```
__asm__ __volatile__("<asm routine>" : output : input : modify);
```

其中：

(1) __asm__表示汇编代码的开始，其后可以跟__volatile__(这是可选项)，其含义是避免 asm 指令被删除、移动或组合。

(2) 在执行代码时，如果不希望汇编语句被 gcc 优化而改变位置，就需要在 asm 符号后添加 volatile 关键词：asm volatile(…)；或者更详细地说明为__asm__ __volatile__(…)；然后就是小括号，括号中的内容是具体的内联汇编指令代码。

(3) "＜asm routine＞"为汇编指令部分，例如，"movl %%cr0,%0\n\t"。数字前加前缀%，如%1、%2 等表示使用寄存器的样板操作数。可以使用的操作数总数取决于具体 CPU 中通用寄存器的数 量，如 Intel 可以有 8 个。

(4) 汇编指令部分中有几个操作数，就说明有几个变量需要与寄存器结合，由 gcc 在编译时根据后面输出部分和输入部分的约束条件进行相应的处理。由于这些样板操作数的前缀使用了“%”，因此，在用到具体的寄存器时就在前面加两个%，如%%cr0。

(5) 输出部分(Output)用以规定对输出变量(目标操作数)如何与寄存器结合的约束(Constraint)，输出部分可以有多个约束，互相以逗号分开。每个约束以=开头，接着用一个字母来表示操作数的类型，然后是关于变量结合的约束。例如，上例中：

```
:"=r" (__dummy)
```

“=r”表示相应的目标操作数(指令部分的%0)可以使用任何一个通用寄存器，并且变量__dummy 存放在这个寄存器中，但如果是：

```
:"=m"(__dummy)
```

“=m”就表示相应的目标操作数是存放在内存单元__dummy 中。表示约束条件的字母很多，表 1-1 给出几个主要的约束字母及其含义。

表 1-1　主要约束字母及含义

字 母	含　　义	字 母	含　　义
m,v,o	内存单元	G	任意
R	任何通用寄存器	a,b,c,d	寄存器 eax/ax/al、ebx/bx/bl、ecx/cx/cl 或 edx/dx/dl
Q	寄存器 eax、ebx、ecx、edx 之一	S,D	寄存器 esi 或 edi
I,h	直接操作数	I	常数(0～31)
E,F	浮点数		

(6) 输入部分(Input)：输入部分与输出部分相似，但没有=。如果输入部分一个操作数所要求使用的寄存器与前面输出部分某个约束所要求的是同一个寄存器，那就把对应操作数的编号(如 1、2 等)放在约束条件中，在后面的例子中会看到这种情况。

(7) 修改部分(Modify)：这部分常常以 Memory 为约束条件，以表示操作完成后内存中的内容已有改变，如果原来某个寄存器的内容来自内存，那么现在内存中这个单元的内容已经改变。注意，指令部分为必选项，而输入部分、输出部分及修改部分为可选项，当输入部分存在，而输出部分不存在时，冒号(:)要保留，当 memory 存在时，三个冒号都要保留，例如：

```
#define __cli() __asm__ __volatile__("cli": : :"memory")
```

下面是一个例子(为方便起见，使用全局变量)：

```
int count=1;
int value=1;
int buf[10];
void main()
{
    asm(
            "cld nt"
            "rep nt"
            "stosl"
        :
        : "c" (count), "a" (value) , "D" (buf[0])
        : "%ecx","%edi"
        );
}
```

得到的主要汇编代码如下：

```
movl count,%ecx
movl value,%eax
movl buf,%edi
#APP
cld
rep
stosl
#NO_APP
```

上述几条汇编指令的功能是向 buf 中写上 count 个 value 值。冒号后的语句指明输入、输出和被改变的寄存器。通过冒号以后的语句，编译器就知道指令需要和改变哪些寄存器，从而可以优化寄存器的分配。其中字符 c(count)指示要把 count 的值放入 ecx 寄存器。类似的还有：

```
a eax
b ebx
c ecx
d edx
S esi
D edi
I 常数值，(0~31)
q,r 动态分配的寄存器
g eax,ebx,ecx,edx 或内存变量
A 把 eax 和 edx 合成一个 64 位的寄存器(use long longs)
```

也可以让 gcc 自己选择合适的寄存器，如下面的例子：

```
asm("leal (%1,%1,4),%0"
    : "=r" (x)
    : "0" (x)
```

```
);
```

得到的主要汇编代码如下：

```
movl x,%eax
#APP
leal (%eax,%eax,4),%eax
#NO_APP
movl %eax,x
```

几点说明如下。

(1) 使用 q 指示编译器从 eax、ebx、ecx、edx 分配寄存器。使用 r 指示编译器从 eax、ebx、ecx、edx、esi、edi 分配寄存器。

(2) 不必把编译器分配的寄存器放入改变的寄存器列表，因为寄存器已经记住了它们。

(3) “=”是标示输出寄存器。

(4) 数字%n 的用法：数字表示的寄存器是按照出现和从左到右的顺序映射到用“r”或“q”请求的寄存器。如果要重用“r”或“q”请求的寄存器的话，就可以使用它们。

(5) 如果强制使用固定的寄存器的话，如不用%1，而用 ebx，则方式如下：

```
asm("leal (%%ebx,%%ebx,4),%0"
    : "=r" (x)
    : "0" (x)
    );
```

注意：要使用两个%，因为一个%的语法已经被%n 用掉了。

2. make 和 makefile

GNU make(简称 make)是一种代码维护工具，在大中型项目中，它将根据程序各个模块的更新情况，自动地维护和生成目标代码。

make 命令执行时，需要一个 makefile(或 makefile)文件，以告诉 make 命令需要怎样去编译和链接程序。首先，我们用一个示例来说明 makefile 的书写规则，以便给大家一个感性认识。这个示例来源于 gnu 的 make 使用手册，在这个示例中，工程有 8 个 c 文件和 3 个头文件，写一个 makefile 来告诉 make 命令如何编译和链接这几个文件。规则如下：

(1) 如果这个工程没有编译过，那么所有 c 文件都要编译并被链接。

(2) 如果这个工程的某几个 c 文件被修改，那么只编译被修改的 c 文件，并链接目标程序。

(3) 如果这个工程的头文件被改变了，那么需要编译引用了这几个头文件的 c 文件，并链接目标程序。

只要我们的 makefile 写得足够好，所有的这一切，只用一个 make 命令就可以完成，make 命令会自动智能地根据当前的文件修改的情况来确定哪些文件需要重编译，从而自己编译所需要的文件和链接目标程序。

在描述 makefile 之前，可粗略地看一看 makefile 的规则。

```
target…: prerequisites…
    command
    ⋮
```

target 是一个目标文件，可以是 object file，也可以是执行文件，还可以是一个标签(label)。prerequisites 就是要生成那个 target 所需要的文件或是目标。command 是 make 需要执行的命令(任意的 shell 命令)。这是一个文件的依赖关系，也就是说，target 这一个或多个的目标文件依赖于 prerequisites 中的文件，其生成规则定义在 command 中。也就是说，prerequisites 中如果有一个以上的文件比 target 文件要新的话，command 所定义的命令就会被执行。这就是 makefile 的规则，也是 makefile 中最核心的内容。

3. gdb 使用

gdb 是功能强大的调试程序，可完成如下调试任务。

(1) 设置断点。

(2) 监视程序变量的值。

(3) 程序的单步(step in/step over)执行。

(4) 显示/修改变量的值。

(5) 显示/修改寄存器。

(6) 查看程序的堆栈情况。

(7) 远程调试。

(8) 调试线程。

在可以使用 gdb 调试程序之前，必须使用-g 或-ggdb 编译选项编译源文件。运行 gdb 调试程序时通常使用如下的命令：

```
gdb progname
```

在 gdb 提示符处输入 help，将列出命令的分类，主要的分类如下。

(1) aliases：命令别名。

(2) breakpoints：断点定义。

(3) data：数据查看。

(4) files：指定并查看文件。

(5) internals：维护命令。

(6) running：程序执行。

(7) stack：调用栈查看。

(8) status：状态查看。

(9) tracepoints：跟踪程序执行。

输入 help 后跟命令的分类名，可获得该类命令的详细清单。gdb 的常用命令如表 1-2 所示。

表 1-2　gdb 的常用命令

名　　称	介　　绍
break FILENAME:NUM	在特定源文件特定行上设置断点
clear FILENAME:NUM	删除设置在特定源文件特定行上的断点
run	运行调试程序
step	单步执行调试程序，不会直接执行函数
next	单步执行调试程序，会直接执行函数

续表

名　　称	介　　绍
backtrace	显示所有的调用栈帧。该命令可用来显示函数的调用顺序
continue	继续执行正在调试的程序
display EXPR	每次程序停止后显示表达式的值,表达式由程序定义的变量组成
file FILENAME	装载指定的可执行文件进行调试
help CMDNAME	显示指定调试命令的帮助信息
info break	显示当前断点列表,包括到达断点处的次数等
info files	显示被调试文件的详细信息
info func	显示被调试程序的所有函数名称
info prog	显示被调试程序的执行状态
info local	显示被调试程序当前函数中的局部变量信息
info var	显示被调试程序的所有全局和静态变量名称
kill	终止正在被调试的程序
list	显示被调试程序的源代码
quit	退出 gdb

下面通过一个有错误的例子程序来介绍 gdb 的使用：

```
/* bugging.c* /
#include<stdio.h>
#include<stdlib.h>

static char buff [256];
static char* string;
int main ()
{
    printf ("Please input a string: ");
    gets (string);
    printf ("\nYour string is: %s\n", string);
}
```

上面这个程序非常简单,其目的是接受用户的输入,然后将用户的输入打印出来。该程序使用了一个未初始化的字符串地址 string,因此,编译并运行之后,将出现 Segment Fault 错误：

```
chy@chyhome-PC:~$gcc  -g  -o bugging bugging.c
chy@chyhome-PC: ~$ ./bugging
Please input a string: asdf
Segmentation fault (core dumped)
```

为了查找该程序中出现的问题,利用 gdb 并按如下步骤进行：

(1) 运行“gdb bugging”,加载 bugging 可执行文件：

```
chy@chyhome-PC: ~$gdb bugging
```

（2）执行装入的 bugging 命令：

```
(gdb) run
```

（3）使用 where 命令查看程序出错的地方：

```
(gdb)where
```

（4）利用 list 命令查看调用 gets 函数附近的代码：

```
(gdb)list
```

（5）在 gdb 中，我们在第 11 行处设置断点，看看是否是在第 11 行出错：

```
(gdb)break 11
```

（6）程序重新运行到第 11 行处停止，这时程序正常，然后执行单步命令 next：

```
(gdb)next
```

（7）程序确实出错，能够导致 gets 函数出错的因素就是变量 string。重新执行测试程，用 print 命令查看 string 的值：

```
(gdb)run
(gdb)print string
(gdb)$ 1= 0x0
```

（8）问题在于 string 指向的是一个无效指针，修改程序，在 10 行和 11 行之间增加一条语句“string＝buff;”，重新编译程序，然后继续运行，将看到正确的程序运行结果。

用 gdb 查看源代码可以用 list 命令，但这不够灵活。也可以用使用-tui 参数，这样进入 gdb 里面后就能直接打开代码查看窗口。其他与代码窗口相关的命令如下：

info win	显示窗口的大小
layout next	切换到下一个布局模式
layout prev	切换到上一个布局模式
layout src	只显示源代码
layout asm	只显示汇编代码
layout split	显示源代码和汇编代码
layout regs	增加寄存器内容显示
focus cmd/src/asm/regs/next/prev	切换当前窗口
refresh	刷新所有窗口
tui reg next	显示下一组寄存器
tui reg system	显示系统寄存器
update	更新源代码窗口和当前执行点
winheight name+/- line	调整 name 窗口的高度
tabset nchar	设置 tab 为 nchar 个字符

4．进一步的相关内容

请同学们在网上搜寻相关资料学习，例如：

gcc tools 相关文档，版本管理软件（CVS、SVN、GIT 等）的使用，等等。

1.2.4 基于硬件模拟器实现源码级调试

1. 安装硬件模拟器 QEMU

1）Linux 运行环境

QEMU 用于模拟一台 x86 计算机，让 ucore 能够运行在 QEMU 上。为了能够正确地编译和安装 qemu，尽量使用最新版本的 qemu（http://wiki.qemu.org/Download），或者 OS FTP 服务器上提供的 qemu 源码 qemu-1.1.0.tar.gz。目前 qemu 能够支持最新的 gcc-4.x 编译器。例如，在 Ubuntu12.04 系统中，默认得版本是 gcc-4.6.x（可以通过 gcc -v 或者 gcc --version 进行查看）。

也可直接使用 ubuntu 中提供的 qemu，只需执行如下命令即可：

```
chy@chyhome-PC: ~$ sudo apt-get install qemu-system
```

也可以采用下面描述的方法对 qemu 进行源码级安装。

2）Linux 环境下的源码级安装过程

（1）获得并应用修改。

编译 qemu 用到的库文件还有 **libsdl1.2-dev** 等。安装命令如下：

```
chy@chyhome-PC: ~$ sudo apt-get install libsdl1.2-dev     //安装库文件 libsdl1.2-dev
```

获得 qemu 的安装包以后，对其进行解压缩（如果格式无法识别，请下载相应的解压缩软件）。

例如，qemu.tar.gz/qemu.tar.bz2 文件，在命令行中可以使用：

```
chy@chyhome-PC: ~$ tar zxvf qemu.tar.gz
```

或者

```
chy@chyhome-PC: ~$ tar jxvf qemu.tar.bz2
```

对 qemu 应用修改：如果实验中使用的 qemu 需要打 patch，应用过程如下：

```
chy@chyhome-PC:~$ ls
qemu.patch qemu
chy@chyhome-PC:~$ cd qemu
chy@chyhome-PC:~/qemu$ patch -p1 -u< ../qemu.patch
```

（2）配置、编译和安装。

编译以及安装 qemu 前需要使用<qemu>（表示 qemu 解压缩路径）下面的 configure 脚本生成相应的配置文件等，而 configure 脚本有较多的参数可供选择，可以通过如下命令进行查看：

```
cd qemu
chy@chyhome-PC: ~/qemu$  ./configure --help
```

实验中可能会用到的命令如下：

```
chy@chyhome-PC: ~/qemu$  ./configure --target-list="i386-softmmu"
                                                  //配置 qemu,可模拟 x86-32 硬件环境
```

```
chy@ chyhome- PC: ~ /qemu$ ./make                              //编译 qemu
chy@ chyhome- PC: ~ /qemu$  ./sudo make install                //安装 qemu
```

注意：版本小于 0.10.0 的 qemu 仅支持 gcc-3.x 版本编译器。但 0.10.x 以上版本的 qemu 已经支持用 gcc-4.x 编译器了。

qemu 执行程序将默认安装到/usr/local/bin 目录下。

如果使用的是默认的安装路径，那么在/usr/local/bin 下面即可看到安装结果：

```
chy@ chy home- PC: ~ /qemu$ ls/usr/Local/bin
qemu- system- i386 qemu- img qemu- nbd…
```

建立符号链接文件 qemu：

```
sudo ln  - s /usr/local/bin/qemu- system- i386  /usr/local/bin/qemu
```

2. 使用硬件模拟器 QEMU

1）运行参数

如果 qemu 使用的是默认/usr/local/bin 安装路径，则在命令行中可以直接使用 qemu 命令运行程序。qemu 运行可以有多参数，格式如下：

```
qemu[options] [disk_image]
```

其中，disk_image 是硬盘镜像文件。

部分参数说明：

'-hda file'/'-hdb file'/'-hdc file'/'-hdd file'

使用 file 作为硬盘 0、1、2、3 镜像。

'-fda file'/'-fdb file'

使用 file 作为软盘镜像，可以使用/dev/fd0 作为 file 来使用主机软盘。

'-cdrom file'

使用 file 作为光盘镜像，可以使用/dev/cdrom 作为 file 来使用主机 cd-rom。

'-boot[a|c|d]'

从软盘(A)、光盘(C)、硬盘启动(D)，默认硬盘启动。

'-snapshot'

写入临时文件而不写回磁盘镜像，可以使用 C-a s 来强制写回。

'-m megs'

设置虚拟内存为 msg MB，默认为 128MB。

'-smp n'

设置为有 n 个 CPU 的 SMP 系统。以 PC 为目标机，最多支持 255 个 CPU。

'-nographic'

禁止使用图形输出。

其他：

可用的主机设备 dev 例如：

vc

虚拟终端。

null

空设备。

/dev/XXX

使用主机的 tty。

file：filename

将输出写入到文件 filename 中。

stdio

标准输入输出。

pipe：pipename

命令管道 pipename。

…

使用 dev 设备的命令如下：

'-serial dev'

重定向虚拟串口到主机设备 dev 中。

'-parallel dev'

重定向虚拟并口到主机设备 dev 中。

'-monitor dev'

重定向 monitor 到主机设备 dev 中。

其他参数：

'-s'

等待 gdb 连接到端口 1234。

'-p port'

改变 gdb 连接端口到 port。

'-S'

在启动时不启动 CPU，需要在 monitor 中输入'c'，才能让 qemu 继续模拟工作。

'-d'

输出日志到 qemu.log 文件。

其他参数说明可以参考网址 http://bellard.org/qemu/qemu-doc.html#SEC15 。其他 qemu 的安装和使用的说明可以参考网址 http://bellard.org/qemu/user-doc.html。

或者在命令行输入 qemu(没有参数)显示帮助。

在实验中，例如 lab1，可能用到的命令如下：

```
chy@chyhome-PC:~$cd  ~/ucore_lab/code/lab1
chy@chyhome-PC:~/ucore_lab/code/lab1$make
chy@chyhome-PC:~/ucore_lab/code/lab1$cd bin
chy@chyhome-PC:~/ucore_lab/code/lab1/bin$qemu -hda ucore.img -parallel stdio      //让 ucore 在 qemu模拟的 x86硬件环境中执行
```

或

```
qemu-S-s-hda ucore.img-monitor stdio                          //用于与 gdb配合进行源码调试
```

2）常用调试命令

执行"qemu-hda acore.img-monitor stdio"可在命令行方式下进入 qemu 的 monitor 子模块对 ucore 操作系统进行监控。

qemu 中 monitor 的常用命令如表 1-3 所示。

表 1-3　monitor 的常用命令

help	查看 qemu 帮助，显示所有支持的命令
q|quit|exit	退出 qemu
stop	停止 qemu
c|cont|continue	连续执行
x/fmt addr xp/fmt addr	显示内存内容，其中 x 为虚地址，xp 为实地址； 参数/fmt i 表示反汇编，默认参数为前一次参数
p|print	计算表达式值并显示，例如，$ reg 表示寄存器结果
memsave addr size file pmemsave addr size file	将内存保存到文件，memsave 为虚地址，pmemsave 为实地址
breakpoint 相关	设置、查看以及删除 breakpoint，PC 执行到 breakpoint，qemu 停止（暂时没有此功能）
watchpoint 相关	设置、查看以及删除 watchpoint，当 watchpoint 地址内容被修改，停止（暂时没有此功能）
s|step	单步一条指令，能够跳过断点执行
r|registers	显示全部寄存器内容
info 相关操作	查询 qemu 支持的关于系统状态信息的操作

其他具体的命令格式以及说明，参见 qemu help 命令帮助。

注意：qemu 默认有 singlestep arg 命令（arg 为参数），该命令为设置单步标志命令。例如，singlestep off 运行结果为禁止单步，singlestep on 结果为允许单步。在允许单步的条件下，使用 cont 命令进行单步操作。例如：

```
(qemu) xp /3i$ pc
0xfffffff0: ljmp      $ 0xf000,$ 0xe05b
0xfffffff5: xor       % bh, (% bx, % si)
0xfffffff7: das
(qemu)singlestep on
(qemu)cont
          0x000fe05b: xor % ax, % ax
```

step 命令为单步命令，即 qemu 执行一步，能够跳过 breakpoint 执行。如果此时使用 cont 命令，则 qemu 运行改为连续执行。

log 命令能够保存 qemu 模拟过程产生的信息（与 qemu 运行参数-d 相同），具体参数可以参考命令帮助。产生的日志信息保存在 /tmp/qemu.log 中，例如，使用 log in_asm 命令以后，运行过程产生的 qemu.log 文件如下：

```
1----------------
2 IN:
3 0xfffffff0:   ljmp   $ 0xf000,$ 0xe05b
4
5----------------
6 IN:
7 0x000fe05b:   xor   % ax,% ax
```

```
 8 0x000fe05d:  out   %al,$0xd
 9 0x000fe05f:  out   %al,$0xda
10 0x000fe061:  mov   $0xc0,%al
11 0x000fe063:  out   %al,$0xd6
12 0x000fe065:  mov   $0x0,%al
13 0x000fe067:  out   %al,$0xd4
```

3. 基于 qemu 内建模式调试 ucore

调试举例：调试 lab1，跟踪 bootmain 函数。

(1) 运行 qemu-S-hdaucore.img-monitor stdio。

(2) 查看 bootblock.asm 得到 bootmain 函数地址为 0x7d60，并插入断点。

(3) 使用命令 c 连续执行到断点。

(4) 使用 xp 命令进行反汇编。

(5) 使用 s 命令进行单步执行。

运行结果如下：

```
chy@chyhome-PC:~/ucorelab/code/lab1$ qemu-S-hda ucore.img-monitor stdio
(qemu)b 0x7d60
insert breakpoint 0x7d60 success!
(qemu)c
    working…
(qemu)
    break:
        0x00007d60:  push  %ebp
(qemu) xp /10i$pc
    0x00007d60:        push      %ebp
    0x00007d61:        mov       %esp,%ebp
    0x00007d63:        push      %esi
    0x00007d64:        push      %ebx
    0x00007d65:        sub       $0x4,%esp
    0x00007d68:        mov       0x7da8,%esi
    0x00007d6e:        mov       $0x0,%ebx
    0x00007d73:        movsbl    (%esi,%ebx,1),%eax
    0x00007d77:        mov       %eax,(%esp,1)
    0x00007d7a:        call      0x7c6c
(qemu)step
        0x00007d61:  mov       %esp,%ebp
(qemu)step
        0x00007d63:  push      %esi
```

4. 结合 gdb 和 qemu 源码级调试 ucore

1) 编译可调试的目标文件

为了使编译出来的代码能够被 gdb 调试，需要在使用 gcc 编译源文件的时候添加参数：-g-gdb 。这样编译出来的目标文件中才会包含可以用于调试器进行调试的相关符号信息。

2) ucore 代码编译

(1) 编译过程：在解压缩后的 ucore 源码包中使用 make 命令即可。例如，lab1 中：

```
chy@chyhome-PC:~/ucore_lab/code/lab1$ make
```

则会在 lab1 目录下的 bin 目录中生成的目标文件为 ucore.img。

(2) 保存修改：使用 diff 命令对修改后的 ucore 代码和 ucore 源码进行比较，比较之前建议使用 make clean 命令清除不必要文件(如果有 ctags 文件，需要手工清除)。

(3) 应用修改：参见 patch 命令说明。

3) 使用远程调试

为了与 qemu 配合进行源代码级别的调试，需要先让 qemu 进入等待 gdb 调试器的接入并且还不能让 qemu 中的 CPU 执行，因此启动 qemu 的时候，需要使用参数"-S"和"-s"这两个参数来做到这一点。在使用了前面提到的参数启动 qemu 之后，qemu 中的 CPU 并不会马上开始执行，这时启动 gdb，然后在 gdb 命令行界面下，使用下面的命令连接到 qemu：

```
(gdb) target remote 127.0.0.1:1234
```

然后输入 c(也就是 continue)命令之后，qemu 会继续执行下去，但是由于 gdb 不知道任何符号信息，并且也没有设下断点，是不能进行源码级的调试的。为了让 gdb 获知符号信息，需要指定调试目标文件，可在 gdb 中使用 file 命令：

```
(gdb)file obj/kernel/kernel.elf
```

之后 gdb 就会载入这个文件中的符号信息了。

通过 gdb 可以对 ucore 代码进行调试，下面以 lab1 中调试 memset 函数为例。

(1) 运行 qemu -S -s -hda ucore.img -monitor stdio。

(2) 运行 gdb 并与 qemu 进行连接。

(3) 设置断点并执行。

(4) qemu 单步调试。

运行过程以及结果如下：

窗口一	窗口二
chy@chyhome-PC:~/ucore_lab/code/lab1$ qemu -S -hda ./bin/ucore.img -s	chy@chyhome-PC:~/ucore_lab/code/lab1$ gdb ./bin/kernel (gdb)target remote:1234 Remote debugging using :1234 0x0000fff0 in ?? () (gdb)file obj/kernel/kernel.elf (gdb)break memset Breakpoint 1 at 0x100d9f: file libs/string.c, line 54. (gdb)run Starting program: /home/chenyu/oscourse/develop/ucore/lab1/bin/kernel Breakpoint 1,memset(s=0x1020fc,c=0 '\000',n=12) at libs/string.c:54 54 return __memset(s, c, n); (gdb)

4）使用 gdb 配置文件

前面讲到，为了进行源码级调试，需要输入较多的东西，这样很麻烦。为了方便，可以将这些命令存在脚本中，并让 gdb 在启动的时候自动载入。

可以创建文件 gdbinit，并输入下面的内容：

```
target remote 127.0.0.1:1234
file obj/kernel/kernel.elf
```

为了让 gdb 在启动时执行这些命令，使用下面的命令启动 gdb：

```
chy@chyhome-PC:~/ucore_lab/code/lab1$ gdb -x gdbinit
```

如果觉得这个命令太长，可以将这个命令存入一个文件中，作为脚本来执行。

另外，如果直接使用上面的命令，那么得到的界面是一个纯命令行的界面，不够直观，如图 1-2 所示。

```
ucore — arm-eabi-gdb — 93×51
arm-eabi-gdb    emulator-arm
43          ldr sp,=abt_stacktop
44          msr cpsr_c,#(DISABLE_IRQ|DISABLE_FIQ|UND_MOD)
45          ldr sp,=und_stacktop
46          msr cpsr_c,#(DISABLE_IRQ|DISABLE_FIQ|SYS_MOD)
47          ldr sp,=sys_stacktop
(gdb) l
48          msr cpsr_c,#(DISABLE_IRQ|DISABLE_FIQ|SVC_MOD)
49
50      #in SVC mode
51      relocated:
52          ldr sp,=bootstacktop
53
54      #if 0
55          # enable the FPU
56          mrc p15, 0, r0, c1, c0, 2
57          orr r0, r0, #0x300000          /* single precision */
(gdb) l
58          orr r0, r0, #0xC00000          /* double precision */
59          mcr p15, 0, r0, c1, c0, 2
60          mov r0, #0x40000000
61          fmxr fpexc,r0
62      #endif
63
64      /* enable swp,flow prediction */
65      #ifdef __MACH_ARM_CORTEX_A9
66          ldr r1, =(1<<11)|(1<<10)
67          MCR p15, 0, r1, c1, c0, 0
(gdb) l
68      #endif
69
70      # now kernel stack is ready , call the first C function
71          b kern_init
72
73      # should never get here
74          spin:
75          b spin
76
77          .data
(gdb) n
=> 0x30938 <kern_entry+4>:      bic     r0, r0, #192    ; 0xc0
   0x3093c <kern_entry+8>:      msr     CPSR_fc, r0
34          bic r0, r0, #(DISABLE_FIQ|DISABLE_IRQ)
(gdb) n
=> 0x3093c <kern_entry+8>:      msr     CPSR_fc, r0
   0x30940 <kern_entry+12>:     msr     CPSR_c, #210    ; 0xd2
35          msr cpsr, r0
(gdb) n
=> 0x30940 <kern_entry+12>:     msr     CPSR_c, #210    ; 0xd2
   0x30944 <kern_entry+16>:     ldr     sp, [pc, #44]   ; 0x30978 <spin+4>
38          msr cpsr_c,#(DISABLE_IRQ|DISABLE_FIQ|IRQ_MOD)
(gdb)
```

图 1-2　命令行界面

如果想获得图 1-3 那样的效果，只需要再加上参数-tui 就行了，例如：

```
chy@chyhome-PC:~$ gdb -tui -x gdbinit
```

5）加载调试目标

如前所述，为了能够让 gdb 识别变量的符号，必须给 gdb 载入符号表等信息。在进行 gdb 本地应用程序调试的时候，由于在指定了执行文件时就已经加载了文件中包含的调试信息，因此不用再使用 gdb 命令专门加载了。但是在使用 qemu 进行远程调试的时候，必须

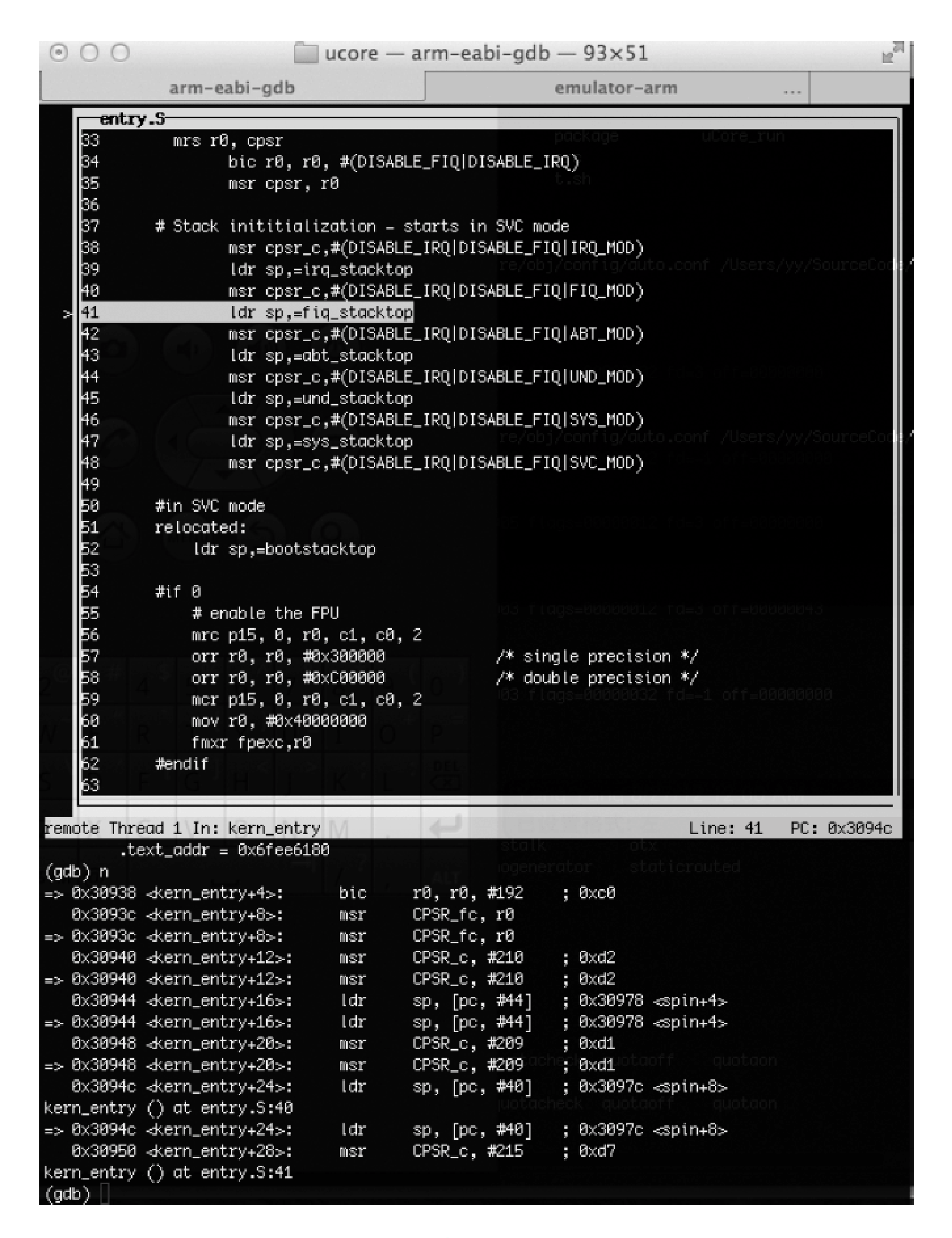


图 1-3

手动加载符号表，也就是在 gdb 中使用 file 命令。

这样加载调试信息都是按照 ELF 文件中制订的虚拟地址进行加载的，这在静态链接的代码中没有任何问题。但是在调试含有动态链接库的代码时，动态链接库的 ELF 执行文件头中指定的加载虚拟地址都是 0，这个地址实际上是不正确的。从操作系统的角度来看，用户态的动态链接库的加载地址都是由操作系统动态分配的，没有一个固定值。然后操作系统再把动态链接库加载到这个地址，并由用户态的库链接器（Linker）把动态链接库中的地址信息重新设置，自此动态链接库才可正常运行。

由于分配地址的动态性，gdb 并不知道这个分配的地址是多少，因此在对这样的动态链接的代码进行调试的时候，需要手动要求 gdb 将调试信息加载到指定地址。

下面，我们要求 gdb 将 linker 加载到 0x6fee6180 这个地址上：

```
(gdb) add-symbol-file test/system/bin/linker 0x6fee6180
```

这样的命令默认是将代码段(.data)的调试信息加载到 0x6fee6180 上，当然，也可以通过-s 这个参数来指定，例如：

```
(gdb)add-symbol-file test/system/bin/linker -s .text 0x6fee6180
```

这样，在执行到 linker 中的代码时 gdb 就能够显示出正确的代码和调试信息出来。这个方法在操作系统中调试动态链接器时特别有用。

6）设定调试目标架构

在调试的时候，有时也许需要调试不是i386保护模式的代码，比如8086实模式的代码，这时需要设定当前使用的架构：

```
(gdb)set arch i8086
```

这个方法在调试不同架构或者说不同模式的代码时还是有点用处的。

1.2.5 了解处理器硬件

要想深入理解ucore，就需要了解支撑ucore运行的硬件环境，即了解处理器体系结构（了解硬件对ucore带来影响）和机器指令集（读懂ucore的汇编）。ucore目前支持的硬件环境是基于Intel 80386以上的计算机系统。更多的硬件相关内容（比如保护模式等）将随着实现ucore的过程逐渐展开介绍。

1. Intel 80386运行模式

80386有四种运行模式：实模式、保护模式、SMM模式和虚拟8086模式，这里仅对涉及ucore的**实模式**和**保护模式**做一个简要介绍。

实模式：80386加电启动后处于实模式运行状态，在这种状态下软件可访问的物理内存空间不能超过1MB，且无法发挥Intel 80386以上级别的32位CPU的4GB内存管理能力。实模式将整个物理内存看成分段的区域，程序代码和数据位于不同区域，操作系统和用户程序并没有区别对待，而且每一个指针都是指向实际的物理地址。这样用户程序的一个指针如果指向了操作系统区域或其他用户程序区域，并修改了内容，那么其后果就很可能是灾难性的。

实模式是为了和8086处理器兼容而设置的。在实模式下，80386处理器就相当于一个快速的8086处理器。80386处理器被复位或加电的时候以实模式启动。这时候处理器中的各寄存器以实模式的初始化值工作。80386处理器在实模式下的存储器寻址方式和8086是一样的，由段寄存器的内容乘以16当做基地址，加上段内的偏移地址形成最终的物理地址，这时候它的32位地址线只使用了低20位，即可访问1MB的物理地址空间。在实模式下，80386处理器不能对内存进行页机制管理，所以指令寻址的地址就是内存中实际的物理地址。在实模式下，所有的段都是可以读、写和执行的。实模式下80386不支持优先级，所有的指令相当于工作在特权级（即优先级0），所以它可以执行所有特权指令，包括读写控制寄存器CR0等。实模式下不支持硬件上的多任务切换。实模式下的中断处理方式和8086处理器相同，也用中断向量表来定位中断服务程序地址。中断向量表的结构也和8086处理器一样，每4B组成一个中断向量，其中包括2B的段地址和2B的偏移地址。

保护模式：实际上，80386就是通过在实模式下初始化控制寄存器、GDTR、LDTR、IDTR与TR等管理寄存器以及页表，然后再通过加载CR0使其中的保护模式使能位置位而进入保护模式的。当80386工作在保护模式下时，其所有的32根地址线都可供寻址，物理寻址空间高达4GB。在保护模式下，支持内存分页机制，提供了对虚拟内存的良好支持。保护模式下80386支持多任务，还支持优先级机制，不同的程序可以运行在不同的优先级上。优先级一共分0～3级，共4级，操作系统运行在最高的优先级0上，应用程序则运行在比较低的级别上；配合良好的检查机制后，既可以在任务间实现数据的安全共享也可以很好

地隔离各个任务。

2. Intel 80386 内存架构

80386 是 32 位的处理器，即可以寻址的物理内存地址空间为 $2^{32}=4GB$。在理解操作系统的过程中，需要用到三个地址空间的概念。地址是访问地址空间的索引。**物理内存地址空间**是处理器提交到总线上用于访问计算机系统中的内存和外设的最终地址。一个计算机系统中只有一个物理地址空间。**线性地址空间**是每个运行的应用程序看到的地址空间，在操作系统的虚存管理之下，每个运行的应用程序都认为自己独享整个计算机系统的地址空间，这样可让多个运行的应用程序之间相互隔离。处理器负责把线性地址转换成物理地址。一个计算机系统中可以有多个线性地址空间（比如一个运行的程序就可以有一个私有的线性地址空间）。线性地址空间的大小与物理地址空间的大小没有必然的连续。**逻辑地址空间**是应用程序直接使用的地址空间。这是由于 80386 中无法禁用段机制，使得逻辑地址一直存在。例如，如下 C 代码片段：

```
int boo=1;
int* foo=&a;
```

这里的 boo 是一个整型变量，foo 变量是一个指向 boo 地址的整型指针变量，foo 中储存的内容就是 boo 的逻辑地址。逻辑地址由一个 16 位的段寄存器和一个 32 位的偏移量构成。foo 中放的就是 32 位的偏移量，而对应的段信息位于段寄存器中。

上述三种地址的关系如下。

（1）分段机制启动、分页机制未启动：逻辑地址→**段机制处理**→线性地址＝物理地址。

（2）分段机制和分页机制都启动：逻辑地址→**段机制处理**→线性地址→**页机制处理**→物理地址。

3. Intel 80386 寄存器

80386 的寄存器可以分为 8 组：通用寄存器、段寄存器、指令指针寄存器、标志寄存器、系统地址寄存器、控制寄存器、调试寄存器、测试寄存器，它们的宽度都是 32 位。一般程序员看到的寄存器包括通用寄存器、段寄存器、指令指针寄存器、标志寄存器。

General Register（通用寄存器）：EAX/EBX/ECX/EDX/ESI/EDI/ESP/EBP 这些寄存器的低 16 位就是 8086 的 AX/BX/CX/DX/SI/DI/SP/BP，对于 AX、BX、CX、DX 这 4 个寄存器来讲，可以单独存取它们的高 8 位和低 8 位（AH、AL、BH、BL、CH、CL、DH、DL）。它们的含义如下：

EAX：累加器。

EBX：基址寄存器。

ECX：计数器。

EDX：数据寄存器。

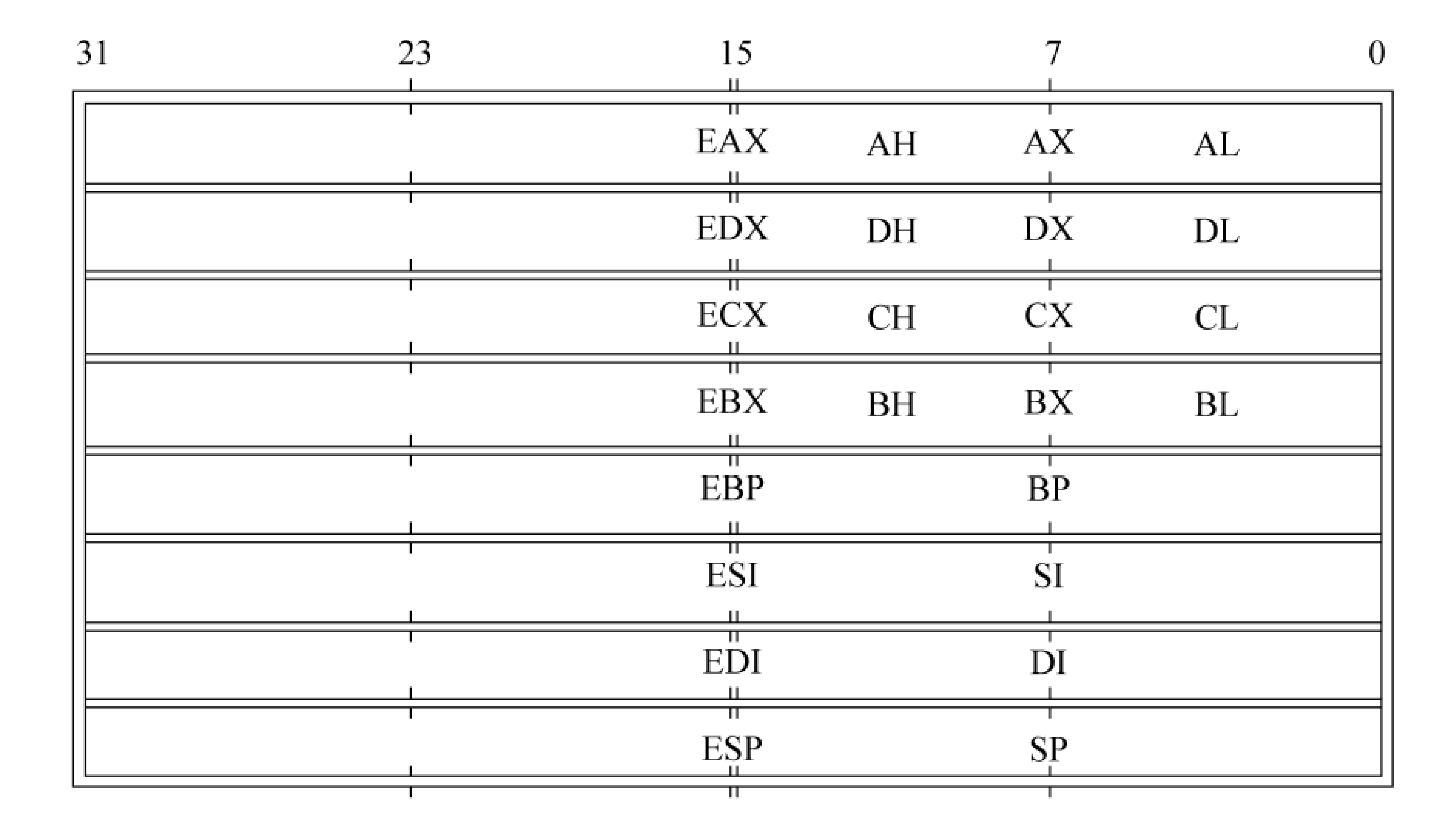
ESI：源地址指针寄存器。

EDI：目的地址指针寄存器。

ESP：堆栈指针寄存器。

EBP：基址指针寄存器。

通用寄存器如下所示。



Segment Register(段寄存器,也称 Segment Selector,段选择符,段选择子):除了 8086 的 4 个段外(CS、DS、ES、SS),80386 还增加了两个段(FS、GS),这些段寄存器都是 16 位的,它们的含义如下。

CS:代码段(Code Segment)。

DS:数据段(Data Segment)。

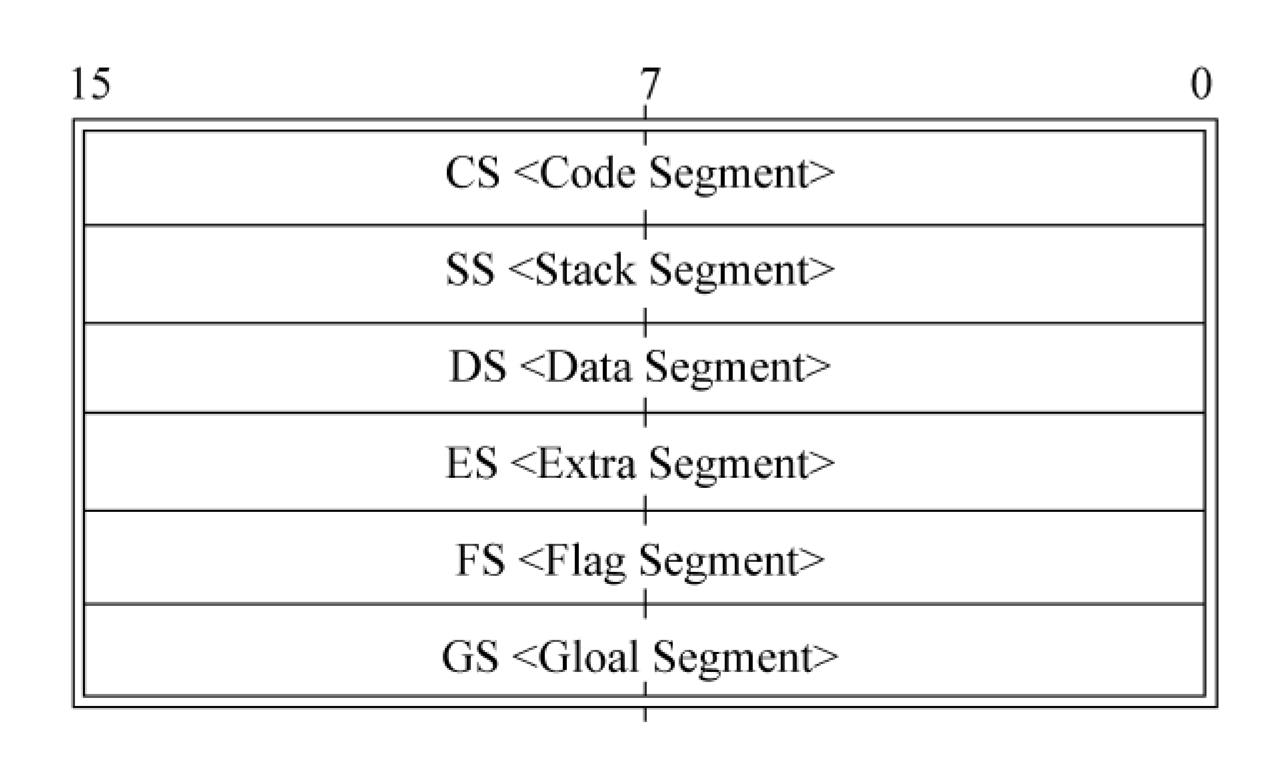
ES:附加段(Extra Segment)。

SS:堆栈段(Stack Segment)。

FS:标志段。

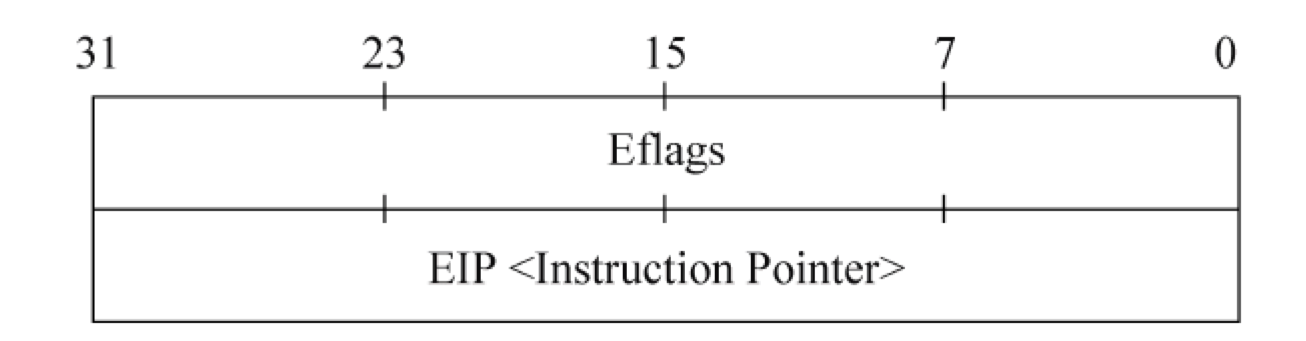
GS:全局段。

段寄存器如下所示。



Instruction Pointer(指令指针寄存器):EIP 的低 16 位就是 8086 的 IP,它存储的是下一条要执行指令的内存地址,在分段地址转换中,表示指令的段内偏移地址。

状态寄存器和指令寄存器如下所示。



Flag Register(标志寄存器):EFLAGS 和 8086 的 16 位标志寄存器相比,增加了 4 个

控制位，这 20 位控制/标志位的位置如图 1-4 所示。

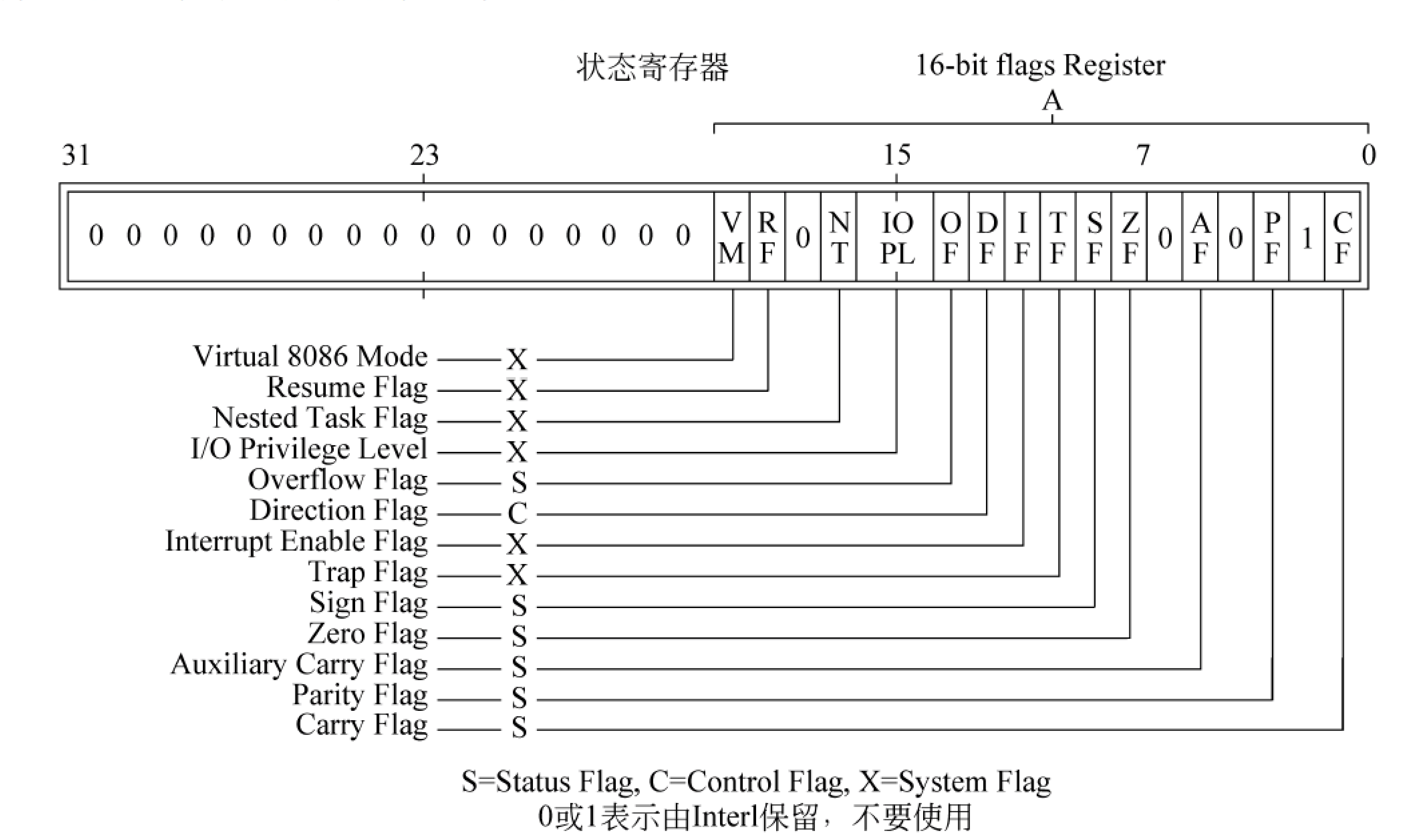


图 1-4　各种标志位

相关的控制/标志位含义如下。

CF(Carry Flag)：进位标志位。

PF(Parity Flag)：奇偶标志位。

AF(Auxiliary Carry Flag)：辅助进位标志位。

ZF(Zero Flag)：零标志位。

SF(Sign Flag)：符号标志位。

IF(Interrupt Enable Flag)：中断允许标志位，由 CLI、STI 两条指令来控制；设置 IF 使 CPU 可识别外部(可屏蔽)中断请求。复位 IF 则禁止中断。IF 对不可屏蔽外部中断和故障中断的识别没有任何作用。

DF(Direction Flag)：向量标志位，由 CLD、STD 两条指令来控制。

OF(Overflow Flag)：溢出标志位。

IOPL(I/O Privilege Level)：I/O 特权级字段，它的宽度为 2 位，它指定了 I/O 指令的特权级。如果当前的特权级别在数值上小于或等于 IOPL，那么 I/O 指令可执行。否则，将发生一个保护性故障中断。

NT(Nested Task Flag)：控制中断返回指令 IRET，它宽度为 1 位。若 NT=0，则用堆栈中保存的值恢复 Eflags、CS 和 EIP 从而实现中断返回；若 NT=1，则通过任务切换实现中断返回。

1.2.6　了解 ucore 编程方法和通用数据结构

1. 面向对象编程方法

在 ucore 设计中采用了一些面向对象编程方法。虽然 C 语言对面向对象编程并没有原生支持，但没有原生支持并不等于不能用 C 语言写面向对象程序。需要注意，我们并不需

要用 C 语言模拟出一个常见 C++ 编译器已经实现的对象模型。如果是那样，还不如直接采用 C++ 编程。

ucore 的面向对象编程方法，目前主要是采用了类似 C++ 的接口(Interface)概念，即让实现细节不同的某类内核子系统(比如物理内存分配器、调度器、文件系统等)有共同的操作方式，这样虽然内存子系统的实现千差万别，但它的访问接口是不变的。这样不同的内核子系统之间就可以灵活地组合在一起，实现风格各异、功能不同的操作系统。接口在 C 语言中，表现为一组函数指针的集合。放在 C++ 中，即为虚表。接口设计的难点是，如果找出各种内核子系统的共性访问/操作模式，从而可以根据访问模式提取出函数指针列表。

例如，对于 ucore 内核中的物理内存管理子系统，首先通过分析内核中其他子系统可能的物理内存管理子系统，明确物理内存管理子系统的访问/操作模式，然后定义 pmm_manager 数据结构(位于 lab2/kern/mm/pmm.h)如下：

```
//pmm_manager is a physical memory management class. A special pmm manager- XXX_pmm_manager
//only needs to implement the methods in pmm_manager class, then XXX_pmm_manager can be used
//by ucore to manage the total physical memory space.
struct pmm_manager {
    //XXX_pmm_manager's name
    const char * name;
    //initialize internal description&management data structure
    //(free block list, number of free block)of XXX_pmm_manager
    void(* init)(void);
    //setup description&management data structcure according to
    //the initial free physical memory space
    void(* init_memmap)(struct Page * base, size_t n);
    //allocate>=n pages, depend on the allocation algorithm
    struct Page * (* alloc_pages)(size_t n);
    //free>=n pages with"base"addr of Page descriptor structures(memlayout.h)
    void (* free_pages)(struct Page * base, size_t n);
    //return the number of free pages
    size_t (* nr_free_pages)(void);
    //check the correctness of XXX_pmm_manager
    void(* check)(void);
};
```

基于此数据结构，我们可以实现不同连续内存分配算法的物理内存管理子系统，而这些物理内存管理子系统需要编写算法，把算法实现在此结构中定义的 init(初始化)、init_memmap(分析空闲物理内存并初始化管理)、alloc_pages(分配物理页)、free_pages(释放物理页)函数指针所对应的函数中。而其他内存子系统需要与物理内存管理子系统交互时，只需调用特定物理内存管理子系统所采用的 pmm_manager 数据结构变量中的函数指针即可。

2. 通用数据结构

双向循环链表

在“数据结构”课程中，如果创建某种数据结构的双循环链表，通常采用的方法是在这个数据结构的类型定义中有专门的成员变量 data，并且加入两个指向该类型的指针 next 和 prev。例如：

```
typedef struct foo {
    ElemType data;
    struct foo * prev;
    struct foo * next;
} foo_t;
```

双向循环链表的特点是尾节点的后继指向首节点，且从任意一个节点出发，沿两个方向的任何一个，都能找到链表中的任意一个节点的 data 数据。由双向循环列表形成的数据链如图 1-5 所示。

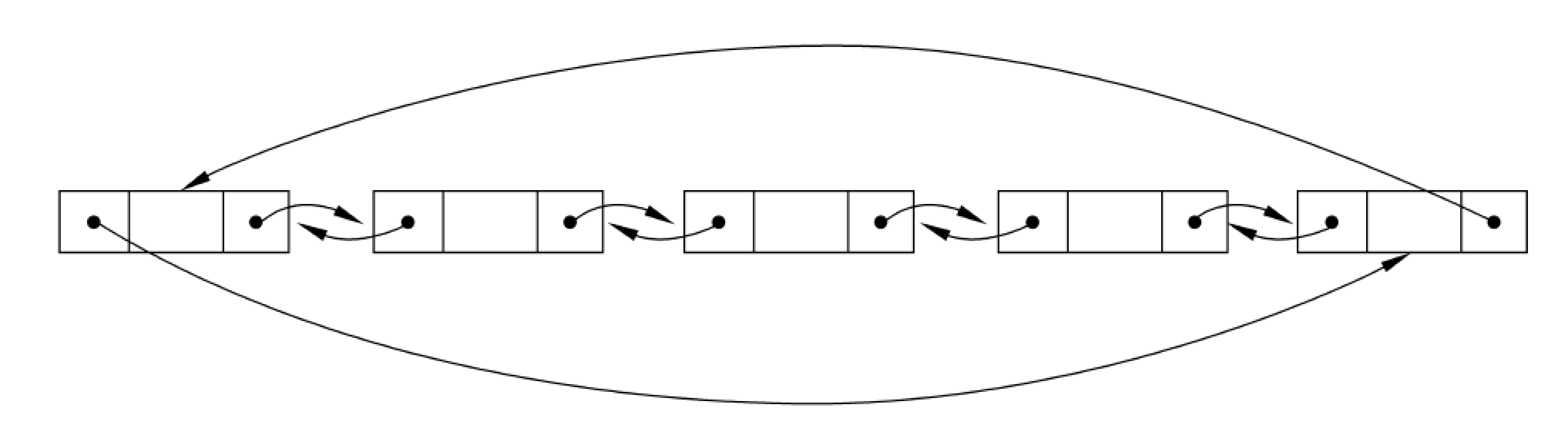


图 1-5 双向循环链表

这种双向循环链表数据结构的一个潜在问题是，虽然链表的基本操作是一致的，但由于每种特定数据结构的类型不一致，需要为每种特定数据结构类型定义针对这个数据结构的特定链表插入、删除等各种操作，这样会导致代码冗余。

在 ucore 内核(从 lab2 开始)中使用了大量的双向循环链表结构来组织数据，包括空闲内存块列表、内存页链表、进程列表、设备链表、文件系统列表等的数据组织(在 libs/list. h 实现)，但其具体实现借鉴了 Linux 内核的双向循环链表实现，与“数据结构”课中的链表数据结构不太一样。下面介绍这一数据结构的设计与操作函数。

ucore 的双向链表结构定义如下：

```
struct list_entry {
    struct list_entry * prev, * next;
};
```

需要注意，ucore 内核的链表节点 list_entry 没有包含传统的 data 数据域，而是在具体的数据结构中包含链表节点。以 lab2 中的空闲内存块列表为例，空闲块链表的头指针定义(位于 lab2/kern/mm/memlayout. h 中)如下：

```
/* free_area_t-
maintains a doubly linked list to record free (unused) pages * /
typedef struct {
    list_entry_t free_list;         //the list header
    unsigned int nr_free;           //number of free pages in this free list
} free_area_t;
```

而每一个空闲块链表节点定义(位于 lab2/kern/mm/memlayout)如下:

```
/**
 * struct Page- Page descriptor structures. Each Page describes one
 * physical page. In kern/mm/pmm.h, you can find lots of useful functions
 * that convert Page to other data types, such as phyical address.
 **/
struct Page {
    atomic_t ref;                   //page frame's reference counter
    ⋮
    list_entry_t page_link;         //free list link
};
```

这样以 free_area_t 结构的数据为双向循环链表的链表头指针,以 Page 结构的数据为双向循环链表的链表节点,就可以形成一个完整的双向循环链表,如图 1-6 所示。

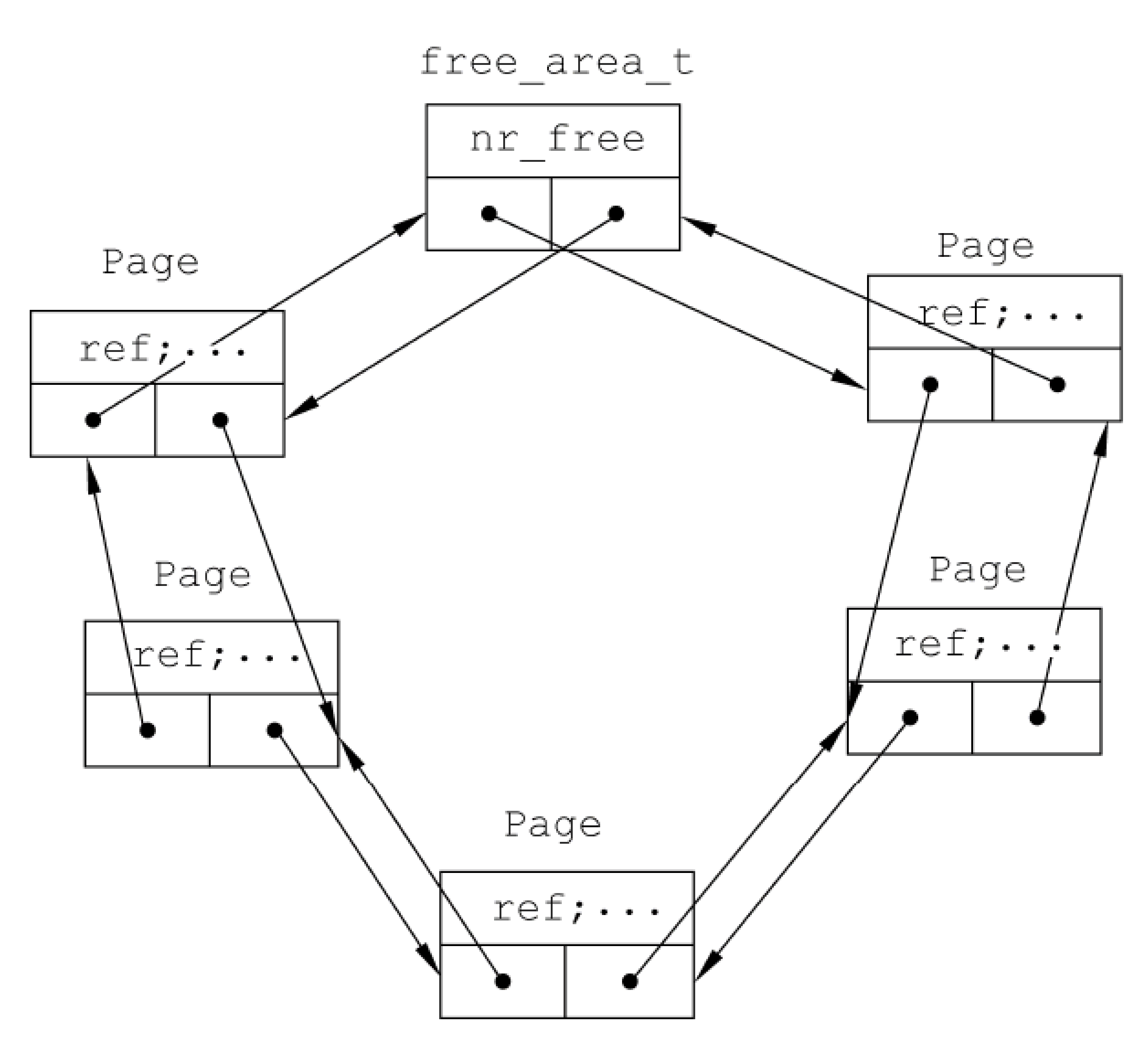


图 1-6　空闲块双向循环链表

从图 1-6 中可以看到,这种通用的双向循环链表结构有个优点,它避免了为每个特定数据结构类型定义针对这个数据结构的特定链表的麻烦,而可以让所有的特定数据结构共享通用的链表操作函数。在实现对空闲块链表的管理过程(参见 lab2/kern/mm/default_pmm.c)中,就大量使用了通用的链表插入、链表删除等操作函数。有关这些链表操作函数的定义如下。

(1) 初始化。

ucore 只定义了链表节点,并没有专门定义链表头,那么一个双向循环链表是如何建立起来的呢?让我们来看看 list_init 这个内联函数(Inline Funciton):

```
static inline void
list_init(list_entry_t* elm) {
    elm->prev=elm->next=elm;
}
```

参见文件 default_pmm.c 的函数 default_init，当调用 list_init(&(free_area.free_list))时，就声明一个名为 free_area.free_list 的链表头，它的 next、prev 指针都初始化为指向自己，这样，我们就有了一个表示空闲内存块链的空链表，而且可以用头指针的 next 是否指向自己来判断此链表是否为空，这就是内联函数 list_empty 的实现。

(2) 插入。

对链表的插入有两种操作，即在表头插入(list_add_after)或在表尾插入(list_add_before)。因为双向循环链表的链表头的 next、prev 分别指向链表中的第一个和最后一个节点，所以，list_add_after 和 list_add_before 的实现区别并不大，实际上 ucore 分别用__list_add(elm, listelm, listelm－＞next)和__list_add(elm, listelm－＞prev, listelm)来实现在表头插入和在表尾插入。__list_add 的实现如下：

```
static inline void
__list_add(list_entry_t * elm,list_entry_t * prev,list_entry_t * next){
    prev->next=next->prev=elm;
    elm->next=next;
    elm->prev=prev;
}
```

从上述实现可以看出，在表头插入是插入 listelm 之后，即插在链表的前端；而在表尾插入是插入 listelm－＞prev 之后，即插在链表的最后。

注：list_add 等于 list_add_after。

(3) 删除。

当需要删除空闲块链表中的 Page 结构的链表节点时，可调用内联函数 list_del，而 list_del 进一步调用了__list_del 来完成具体的删除操作。其实现如下：

```
static inline void
list_del(list_entry_t * listelm) {
    __list_del(listelm->prev,listelm->next);
}

static inline void
__list_del(list_entry_t * prev,list_entry_t * next) {
    prev->next=next;
    next->prev=prev;
}
```

如果要确保被删除的节点 listelm 不再指向链表中的其他节点，这可以通过调用 list_init 函数来让 listelm 的 prev、next 指针分别指向自身，即将节点置为空链状态。这可以通过 list_del_init 函数来完成。

(4) 访问链表节点所在的宿主数据结构。

通过上面的描述可知，list_entry_t 通用双向循环链表中仅保存了某特定数据结构中链表节点成员变量的地址，那么如何通过这个链表节点成员变量访问它的所有者(即某特定数据结构的变量)呢？Linux 为此提供了针对数据结构 XXX 的 le2XXX(le, member)的宏，其中 le 即 list entry 的简称，是指向数据结构 XXX 中 list_entry_t 成员变量的指针，也就是存

储在双向循环链表中的节点地址值，member 则是 XXX 数据类型中包含的链表节点的成员变量。例如，我们要遍历访问空闲块链表中所有节点所在的基于 Page 数据结构的变量，则可以采用如下编程方式(基于 lab2/kern/mm/default_pmm.c)：

```
//free_area 是空闲块管理结构,free_area.free_list 是空闲块链表头
free_area_t free_area;
list_entry_t * le= &free_area.free_list;                //le 是空闲块链表头指针
while((le= list_next(le))!= &free_area.free_list) {     //从第一个节点开始遍历
    struct Page * p= le2page(le,page_link);             //获取节点所在基于 Page 数据结构的变量
    ⋮
}
```

le2page 宏(定义位于 lab2/kern/mm/memlayout.h)的使用相当简单：

```
//convert list entry to page
#define le2page(le,member)
    \to_struct((le),struct Page,member)
```

相比之下，它的实现用到的 to_struct 宏和 offsetof 宏(定义位于 lab2/libs/defs.h)则有一些难懂：

```
/* Return the offset of 'member' relative to the beginning of a struct type * /
#define offsetof(type, member)
     ((size_t)(&((type * )0)->member))


/**
 * to_struct- get the struct from a ptr
 * @ptr:     a struct pointer of member
 * @type:    the type of the struct this is embedded in
 * @member: the name of the member within the struct
 **/
#define to_struct(ptr, type, member)
     ((type * )((char * )(ptr)-offsetof(type, member)))
```

这里采用了一个利用 gcc 编译器技术的技巧，即先求得数据结构的成员变量在本宿主数据结构中的偏移量，然后根据成员变量的地址反过来得出属主数据结构的变量的地址。

让我们首先来看 offsetof 宏，size_t 最终定义与 CPU 体系结构相关，本实验都采用 Intel x86-32 CPU，故 szie_t 等价于 unsigned int。((type *)0)－＞member 的设计含义是什么？其实这是为了求得数据结构的成员变量在本宿主数据结构中的偏移量。为了达到这个目标，首先将 0 地址强制转换为 type 数据结构(比如 struct Page)的指针，再访问到 type 数据结构中的 member 成员(比如 page_link)的地址，即是 type 数据结构中 member 成员相对于数据结构变量的偏移量。在 offsetof 宏中，这个 member 成员的地址(即 &((type *)0)－＞member))实际上就是 type 数据结构中 member 成员相对于数据结构变量的偏移量。给定一个结构，offsetof(type,member)是一个常量，to_struct 宏正是利用这个不变的

偏移量来求得链表数据项的变量地址。接下来再分析一下 to_struct 宏，可以发现 to_struct 宏中用到的 ptr 变量是链表节点的地址，把它减去 offsetof 宏所获得的数据结构内偏移量，即可得到包含链表节点的属主数据结构的变量的地址。

第 2 章　实验 1：系统软件启动过程

2.1　实 验 目 的

操作系统是一个软件，它也需要通过某种机制加载并运行。可以通过另外一个更加简单的软件——bootloader 来完成这些工作，即需要一个能够切换到 x86 的保护模式并显示字符的 bootloader，为启动操作系统 ucore 做准备。lab1 提供了一个非常小的 bootloader 和 ucore OS，整个 bootloader 执行代码小于 512B，这样才能放到硬盘的主引导扇区中。通过分析和实现这个 bootloader 和 ucore OS，读者可以了解到如下内容。

(1) 基于分段机制的存储管理。

(2) 设备管理的基本概念。

(3) PC 启动 bootloader 的过程。

(4) bootloader 的文件组成。

(5) 编译运行 bootloader 的过程。

(6) 调试 bootloader 的方法。

(7) ucore OS 的启动过程。

(8) 在汇编级了解栈的结构和处理过程。

(9) 中断处理机制。

(10) 通过串口/并口/CGA 输出字符的方法。

2.2　实 验 内 容

lab1 中包含一个 bootloader 和一个 OS。这个 bootloader 可以切换到 x86 保护模式，能够读磁盘并加载 ELF 执行文件格式，并显示字符。lab1 中的 OS 只是一个可以处理时钟中断和显示字符的幼儿园级别的 OS。

2.2.1　练习

练习 1：理解通过 make 生成执行文件的过程(要求在报告中写出对下述问题的回答)。

在此练习中，大家需要通过静态分析代码来了解如下内容。

(1) 操作系统镜像文件 ucore.img 是如何一步一步生成的(需要比较详细地解释 Makefile 中每一条相关命令和命令参数的含义，以及说明命令导致的结果)？

(2) 一个被系统认为是符合规范的硬盘主引导扇区的特征是什么？

补充材料：如何调试 Makefile?

当执行 make 时，一般只会显示输出，不会显示 make 到底执行了哪些命令。

如想了解 make 执行了哪些命令，可以执行：

```
$ make "V="
```

要获取更多有关 make 的信息，可上网查询，并请执行：

```
$ man make
```

练习 2：使用 qemu 执行并调试 lab1 中的软件（要求在报告中简要写出练习过程）。

为了熟悉使用 qemu 和 gdb 进行的调试工作，我们进行如下小练习。

（1）从 CPU 加电后执行的第一条指令开始，单步跟踪 BIOS。

（2）在初始化位置 0x7c00 设置实地址断点，测试断点正常。

（3）从 0x7c00 开始跟踪代码运行，将单步跟踪反汇编得到的代码与 bootasm. S 和 bootblock. asm 进行比较。

（4）自己找一个 bootloader 或内核中的代码位置，设置断点并进行测试。

提示：参考附录“启动后第一条执行的指令”。

补充材料：

我们主要通过硬件模拟器 qemu 来进行各种实验，在实验的过程中可能会遇上各种各样的问题，调试是必要的。qemu 支持使用 gdb 进行强大而方便的调试，所以用好 qemu 和 gdb 是完成各种实验的基本条件。

默认的 gdb 需要进行一些额外的配置才能进行 qemu 的调试任务。qemu 和 gdb 之间使用网络端口 1234 进行通信。在打开 qemu 进行模拟之后，执行 gdb 并输入

```
target remote localhost:1234
```

即可连接 qemu，此时 qemu 会进入停止状态，听从 gdb 的命令。

另外，可能需要 qemu 在一开始便进入等待模式，则不再使用 make qemu 开始系统的运行，而使用 make debug 来完成这项工作。这样 qemu 便不会在 gdb 尚未连接的时候擅自运行了。

BIOS 首先运行在 16 位实模式下，第一条指令是 ljmp，执行这条指令后会跳到另外一个地方。gdb 默认是 32 位线性地址模式，调试 BIOS 的 16 位代码（段地址）需要手动计算地址，计算公式如下：

```
Linear Addr= (cs<<4)+ip
```

如果 CS＝0xf000，EIP＝0xe05b，则 Linear Address＝0xfe05b。

另外，为了正确反汇编 16 位指令，在 gdb 中执行

```
(gdb)set architecture i8086
(gdb)x/16i 0xfe05b
0xfe05b: cmpl $0x0,%cs:-0x2f2c
0xfe062: jne 0xfc792
⋮
```

（1）gdb 的地址断点。

在 gdb 命令行中，使用 b ＊[地址]便可以在指定内存地址设置断点，当 qemu 中的

CPU 执行到指定地址时，便会将控制权交给 gdb。

（2）关于代码的反汇编。

有可能 gdb 无法正确获取当前 qemu 执行的汇编指令，通过如下配置可以在每次 gdb 命令行前强制反汇编当前的指令，在 gdb 命令行或配置文件中添加：

```
define hook-stop
x/i$pc
end
```

即可。

（3）gdb 的单步命令。

在 gdb 中，由 next、nexti、step、stepi 等指令来单步调试程序，它们的功能各不相同，区别在于单步的"跨度"上。

next：单步到程序源代码的下一行，不进入函数。

nexti：单步一条机器指令，不进入函数。

step：单步到下一个不同的源代码行(包括进入函数)。

stepi：单步一条机器指令。

练习 3：分析 bootloader 进入保护模式的过程(要求在报告中进行分析)。

BIOS 将通过读取硬盘主引导扇区到内存，并转跳到对应内存中的位置执行 bootloader。请分析 bootloader 是如何完成从实模式进入保护模式的。

提示：需要阅读 2.3.2 节中"保护模式和分段机制"和 lab1/boot/bootasm.S 源码，了解如何从实模式切换到保护模式。

练习 4：分析 bootloader 加载 ELF 格式的 OS 的过程。(要求在报告中写出分析)。

通过阅读 bootmain.c，了解 bootloader 如何加载 ELF 文件。通过分析源代码和通过 qemu 来运行并调试 bootloader&OS。

（1）bootloader 是如何读取硬盘扇区的？

（2）bootloader 是如何加载 ELF 格式的 OS 的？

提示：可阅读 2.3.2 节中的"硬盘访问概述"和"ELF 执行文件格式概述"。

练习 5：实现函数调用堆栈跟踪函数（需要编程）。

我们需要在 lab1 中实现 kdebug.c 中的函数 print_stackframe，可以通过函数 print_stackframe 来跟踪函数调用堆栈中记录的返回地址。如果能够正确实现此函数，可在 lab1 中执行 make qemu 后，在 qemu 模拟器中得到类似如下的输出：

```
⋮
ebp:0x00007b28 eip:0x00100992 args:0x00010094 0x00010094 0x00007b58 0x00100096
    kern/debug/kdebug.c:305: print_stackframe+22
ebp:0x00007b38 eip:0x00100c79 args:0x00000000 0x00000000 0x00000000 0x00007ba8
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b58 eip:0x00100096 args:0x00000000 0x00007b80 0xffff0000 0x00007b84
    kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b78 eip:0x001000bf args:0x00000000 0xffff0000 0x00007ba4 0x00000029
    kern/init/init.c:53: grade_backtrace1+38
```

```
ebp:0x00007b98 eip:0x001000dd args:0x00000000 0x00100000 0xffff0000 0x0000001d
    kern/init/init.c:58: grade_backtrace0+ 23
ebp:0x00007bb8 eip:0x00100102 args:0x0010353c 0x00103520 0x00001308 0x00000000
    kern/init/init.c:63: grade_backtrace+ 34
ebp:0x00007be8 eip:0x00100059 args:0x00000000 0x00000000 0x00000000 0x00007c53
    kern/init/init.c:28: kern_init+ 88
ebp:0x00007bf8 eip:0x00007d73 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
    <unknow> :-- 0x00007d72-
⋮
```

请完成实验，看看输出是否与上述显示一致，并解释最后一行各个数值的含义。

提示：可阅读 2.3.3 节中的"函数堆栈"，了解编译器是如何建立函数调用关系的。在完成 lab1 编译后，查看 lab1/obj/bootblock.asm，了解 bootloader 源码与机器码的语句和地址等的对应关系；查看 lab1/obj/kernel.asm，了解 ucore OS 源码与机器码的语句和地址等的对应关系。

要求完成函数 kern/debug/kdebug.c::print_stackframe 的实现，提交改进后源代码包(可以编译执行)，并在实验报告中简要说明实现过程，并写出对上述问题的回答。

补充材料

显示完整的栈结构需要解析内核文件中的调试符号，这较为复杂和烦琐。代码中有一些辅助函数可以使用。例如，可以通过调用 print_debuginfo 函数完成查找对应函数名并打印至屏幕的功能。具体可以参见 kdebug.c 代码中的注释。

练习 6：完善中断初始化和处理（需要编程）。

请完成编码工作并回答如下问题。

（1）中断向量表中一个表项占多少字节？其中哪几位代表中断处理代码的入口？

（2）请编程完善 kern/trap/trap.c 中对中断向量表进行初始化的函数 idt_init。在 idt_init 函数中，依次对所有中断入口进行初始化。使用 mmu.h 中的 SETGATE 宏，填充 idt 数组内容。注意除了系统调用中断(T_SYSCALL)以外，其他中断均使用中断门描述符，权限为内核态权限；而系统调用中断使用异常，权限为陷阱门描述符。每个中断的入口由 tools/vectors.c 生成，使用 trap.c 中声明的 vectors 数组即可。

（3）请编程完善 trap.c 中的中断处理函数 trap，在对时钟中断进行处理的部分，请填写 trap 函数中处理时钟中断的部分，使操作系统每遇到 100 次时钟中断后，调用 print_ticks 子程序，向屏幕上打印一行文字"100 ticks"。

要求完成问题(2)和问题(3)提出的相关函数实现，提交改进后的源代码包(可以编译执行)，并在实验报告中简要说明实现过程，并写出对问题 1 的回答。完成问题(2)和问题(3)要求的部分代码后，运行整个系统，可以看到大约每 1s 会输出一次"100 ticks"，而按下的键也会在屏幕上显示。

提示：可阅读 2.3.3 节中的"中断与异常"。

扩展练习 Challenge(需要编程)

扩展 proj4，增加 syscall 功能，即增加一用户态函数(可执行一特定系统调用：获得时钟计数值)，当内核初始完毕后，可从内核态返回到用户态的函数，而用户态的函数又通过系统

调用得到内核态的服务(通过网络查询所需信息,可找老师咨询。需写出详细的设计和分析报告。

提示:规范一下 challenge 的流程。

kern_init 调用 switch_test,该函数如下:

```
static void
switch_test(void) {
    print_cur_status();                    //print 当前 cs/ss/ds 等寄存器状态
    cprintf("+++switch to  user  mode+++\n");
    switch_to_user();                      //switch to user mode
    print_cur_status();
    cprintf("+++switch to kernel mode+++\n");
    switch_to_kernel();                    // switch to kernel mode
    print_cur_status();
}
```

switch_to_* 函数建议通过中断处理的方式实现。主要要完成的代码是在 trap 里面处理 T_SWITCH_TO* 中断,并设置好返回的状态。

在 lab1 里面完成代码以后,执行 make grade 应该能够评测结果是否正确。

2.2.2 项目组成

lab1 的整体目录结构如图 2-1 所示。

其中一些比较重要的文件说明如下。

1. bootloader 部分

(1) boot/bootasm.S :定义并实现 bootloader 最先执行的函数 start,此函数进行了一定的初始化,完成了从实模式到保护模式的转换,并调用 bootmain.c 中的 bootmain 函数。

(2) boot/bootmain.c:定义并实现了 bootmain 函数实现了通过屏幕、串口和并口显示字符串。bootmain 函数加载 ucore 操作系统到内存,然后跳转到 ucore 的入口处执行。

(3) boot/asm.h:是 bootasm.S 汇编文件所需要的头文件,主要是一些与 x86 保护模式的段访问方式相关的宏定义。

2. ucore 操作系统部分

1) 系统初始化部分

kern/init/init.c:ucore 操作系统的初始化启动代码。

2) 内存管理部分

(1) kern/mm/memlayout.h:ucore 操作系统有关段管理(段描述符编号、段号等)的一些宏定义。

(2) kern/mm/mmu.h:ucore 操作系统有关 x86 MMU 等硬件相关的定义,包括 Eflags 寄存器中各位的含义,应用/系统段类型,中断门描述符定义,段描述符定义,任务状态段定义,NULL 段声明的宏 SEG_NULL,特定段声明的宏 SEG,设置中断门描述符的宏 SETGATE(在练习 6 中会用到)。

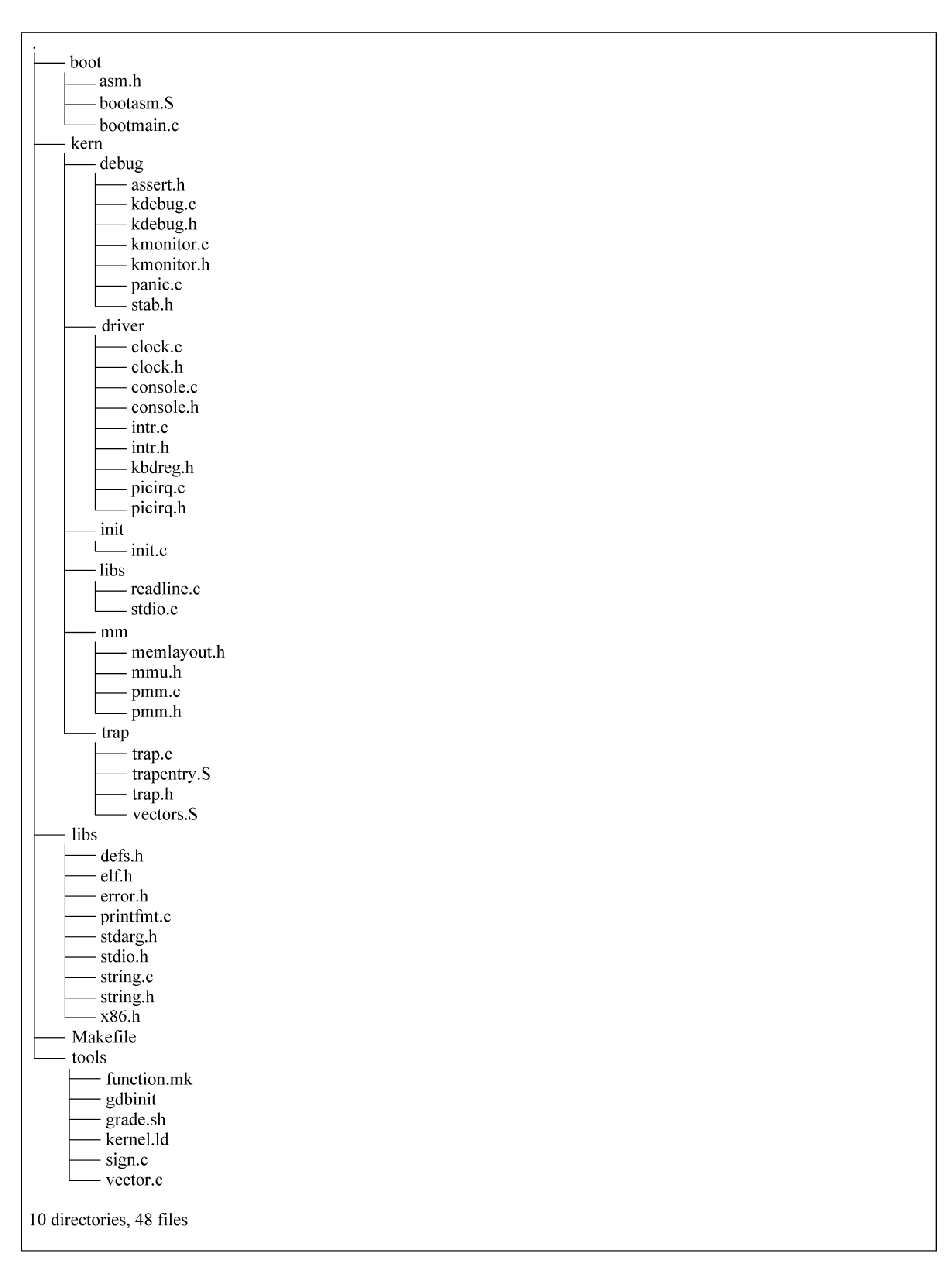

.
boot
asm.h
bootasm.S
bootmain.c
kern
debug
assert.h
kdebug.c
kdebug.h
kmonitor.c
kmonitor.h
panic.c
stab.h
driver
clock.c
clock.h
console.c
console.h
intr.c
intr.h
kbdreg.h
picirq.c
picirq.h
init
init.c
libs
readline.c
stdio.c
mm
memlayout.h
mmu.h
pmm.c
pmm.h
trap
trap.c
trapentry.S
trap.h
vectors.S
libs
defs.h
elf.h
error.h
printfmt.c
stdarg.h
stdio.h
string.c
string.h
x86.h
Makefile
tools
function.mk
gdbinit
grade.sh
kernel.ld
sign.c
vector.c
10 directories, 48 files

图 2-1　目录结构图

(3) kern/mm/pmm.[ch]：设定 ucore 操作系统在段机制中要用到的全局变量：任务状态段 ts，全局描述符表 gdt[]，加载全局描述符表寄存器的函数 lgdt，临时的内核栈 stack0；以及对全局描述符表和任务状态段的初始化函数 gdt_init。

3）外设驱动部分

(1) kern/driver/intr.[ch]：实现了通过设置 CPU 的 Eflags 来屏蔽和使能中断的函数。

(2) kern/driver/picirq.[ch]：实现了对中断控制器 8259A 的初始化和使能操作。

(3) kern/driver/clock.[ch]：实现了对时钟控制器 8253 的初始化操作。

(4) kern/driver/console.[ch]：实现了对串口和键盘的中断方式的处理操作。

4）中断处理部分

(1) kern/trap/vectors.S：包括 256 个中断服务例程的入口地址和第一步初步处理实现。注意，此文件是由 tools/vector.c 在编译 ucore 期间动态生成的。

(2) kern/trap/trapentry.S：紧接着第一步初步处理后，进一步完成第二步初步处理；并且又恢复中断上下文的处理，即中断处理完毕后的返回准备工作。

(3) kern/trap/trap.[ch]：紧接着第二步初步处理后，继续完成具体的各种中断处理操作。

5）内核调试部分

(1) kern/debug/kdebug.[ch]：提供源码和二进制对应关系的查询功能，用于显示调用栈关系。其中，补全 print_stackframe 函数是需要完成的练习，其他实现部分不必深究。

(2) kern/debug/kmonitor.[ch]：实现提供动态分析命令的 kernel monitor，便于在 ucore 出现 bug 或问题后，能够进入 kernel monitor 中，查看当前调用关系。实现部分不必深究。

(3) kern/debug/panic.c | assert.h：提供了 panic 函数和 assert 宏，便于在发现错误后，调用 kernel monitor。大家可在编程实验中充分利用 assert 宏和 panic 函数，提高查找错误的效率。

3. 公共库部分

(1) libs/defs.h：包含一些无符号整型的缩写定义。

(2) libs/x86.h：一些用 GNU C 嵌入式汇编实现的 C 函数(由于使用了 inline 关键字，所以可以理解为宏)。

4. 工具部分

(1) Makefile 和 function.mk：指导 make 完成整个软件项目的编译、清除等工作。

(2) sign.c：一个 C 语言小程序，是辅助工具，用于生成一个符合规范的硬盘主引导扇区。

(3) tools/vector.c：生成 vectors.S，此文件包含了中断向量处理的统一实现。

首先下载 lab1.tar.bz2，然后解压 lab1.tar.bz2。在 lab1 目录下执行 make，可以生成 ucore.img(生成于 bin 目录下)。ucore.img 是一个包含了 bootloader 或 OS 的硬盘镜像，通过执行如下命令可在硬件虚拟环境 qemu 中运行 bootloader 或 OS：

```
$ make qemu
```

2.3 从机器启动到操作系统运行的过程

2.3.1 BIOS 启动过程

当计算机加电后，一般不直接运行操作系统，而是执行系统初始化软件完成基本 I/O 初始化和引导加载。简单地说，系统初始化软件就是在操作系统内核运行之前运行的一段小软件。通过这段小软件，可以初始化硬件设备、建立系统的内存空间映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。最终引导加载程序把操作系统内核映像加载到 RAM 中，并将系统控制权传递给它。

对于绝大多数计算机系统而言，操作系统和应用软件是存放在磁盘(硬盘/软盘)、光盘、EPROM、ROM、Flash 等可在断电后继续保存数据的存储介质上。计算机启动后，CPU 一开始会到一个特定的地址开始执行指令，这个特定的地址存放了系统初始化软件，负责完成计算机基本的 I/O 初始化，这是系统加电后运行的第一段软件代码。对于 Intel 80386 的体系结构而言，PC 中的系统初始化软件由 BIOS (Basic Input Output System，即基本输入输出系统，其本质是一个固化在主板 Flash/CMOS 上的软件)和位于软盘/硬盘引导扇区中的 OS Boot Loader(在 ucore 中的 bootasm. S 和 bootmain. c)一起组成。BIOS 实际上是被固化在计算机 ROM(只读存储器)芯片上的一个特殊的软件，为上层软件提供最底层的、最直接的硬件控制与支持。更形象地说，BIOS 就是计算机硬件与上层软件程序之间的一个“桥梁”，负责访问和控制硬件。

以 Intel 80386 为例，计算机加电后，CPU 从物理地址 0xFFFFFFF0(由初始化的 CS:EIP 确定，此时 CS 和 IP 的值分别是 0xF000 和 0xFFF0))开始执行。在 0xFFFFFFF0 这里只是存放了一条跳转指令，通过跳转指令跳到 BIOS 例行程序起始点。BIOS 完成计算机硬件自检和初始化后，会选择一个启动设备(例如硬盘、光盘等)，并且读取该设备的第一扇区(即主引导扇区或启动扇区)到内存一个特定的地址 0x7c00 处，然后 CPU 控制权会转移到那个地址继续执行。至此 BIOS 的初始化工作做完了，进一步的工作交给了 ucore 的 bootloader。

2.3.2 bootloader 启动过程

BIOS 将通过读取硬盘主引导扇区到内存，并转跳到对应内存中的位置执行 bootloader。bootloader 完成的工作包括如下。

(1) 切换到保护模式，启用分段机制。

(2) 读取磁盘中 ELF 执行文件格式的 ucore 操作系统到内存。

(3) 显示字符串信息。

(4) 把控制权交给 ucore 操作系统。

对应的实现文件为 lab1 中的 boot 目录下的三个文件——asm. h、bootasm. S 和 bootmain. c。下面从原理上介绍完成上述工作的计算机系统硬件和软件背景知识。

1. 保护模式和分段机制

为何要了解 Intel 80386 的保护模式和分段机制？首先，我们知道 Intel 80386 只有在进

入保护模式后，才能充分发挥其强大的功能，提供更好的保护机制和更大的寻址空间，否则仅仅是一个快速的 8086 而已。没有一定的保护机制，任何一个应用软件都可以任意访问所有的计算机资源，这样也就无从谈起操作系统设计了，且 Intel 80386 的分段机制一直存在，无法屏蔽或避免。其次，在 bootloader 的设计中，涉及了从实模式到保护模式的处理，我们的操作系统功能(比如分页机制)是建立在 Intel 80386 的保护模式上来设计的。如果不了解保护模式和分段机制，则面向 Intel 80386 体系结构的操作系统设计实际上是一个空中楼阁。

注意：虽然大家学习过 x86 汇编，对 x86 硬件架构有一定了解，但对 x86 保护模式和 x86 系统编程可能了解不够。为了能够清楚了解各个实验中汇编代码的含义，建议大家阅读如下参考资料。

(1) 可先回顾一下 lab0-manual 中的"了解处理器硬件"一节的内容。

(2)《Intel 80386 Reference Programmers Manual-i386》：第 4、6、9、10 章。在后续实验中，还可以进一步阅读第 5、7、8 等章节。

1) 实模式

在 bootloader 接手 BIOS 的工作后，当前的 PC 系统处于实模式(16 位模式)运行状态，在这种状态下软件可访问的物理内存空间不能超过 1MB，且无法发挥 Intel 80386 以上级别的 32 位 CPU 的 4GB 内存管理能力。

实模式将整个物理内存看成分段的区域，程序代码和数据位于不同区域，操作系统和用户程序并没有区别对待，而且每一个指针都是指向实际的物理地址。这样，用户程序的一个指针如果指向了操作系统区域或其他用户程序区域，并修改了内容，那么其后果就很可能是灾难性的。通过修改 A20 地址线可以完成从实模式到保护模式的转换。有关 A20 的进一步信息可参考本章的附录 A"关于 A20 Gate"。

2) 保护模式

只有在保护模式下，80386 的全部 32 根地址线才有效，可寻址高达 4GB 的线性地址空间和物理地址空间，可访问 64TB(有 2^{14} 个段，每个段最大空间为 2^{32} B)的逻辑地址空间，可采用分段存储管理机制和分页存储管理机制。这不仅为存储共享和保护提供了硬件支持，而且为实现虚拟存储提供了硬件支持。通过提供 4 个特权级和完善的特权检查机制，既能实现资源共享又能保证代码数据的安全及任务的隔离。

3) 分段存储管理机制

只有在保护模式下才能使用分段存储管理机制。分段机制将内存划分成以起始地址和长度限制这两个二维参数表示的内存块，这些内存块就称为段(Segment)。编译器把源程序编译成执行程序时用到的代码段、数据段、堆和栈等概念在这里可以与段联系起来，两者在含义上是一致的。

分段机制涉及 4 个关键内容：逻辑地址、段描述符(描述段的属性)、段描述符表(包含多个段描述符的"数组")、段选择子(段寄存器，用于定位段描述符表中表项的索引)。转换逻辑地址(Logical Address，应用程序员看到的地址)到物理地址(Physical Address，实际的物理内存地址)分以下两步。

(1) 分段地址转换：CPU 把逻辑地址(由段选择子 Selector 和段偏移 Offset 组成)中的段选择子的内容作为段描述符表的索引，找到表中对应的段描述符，然后把段描述符中保存

的段基址加上段偏移值，形成线性地址(Linear Address)。如果不启动分页存储管理机制，则线性地址等于物理地址。

(2) 分页地址转换，这一步中把线性地址转换为物理地址(这一步是可选的，由操作系统决定是否需要，在后续实验中会涉及)。

上述转换过程对于应用程序员来说是不可见的。**线性地址空间**由一维的线性地址构成，线性地址空间和物理地址空间对等。线性地址长 32 位，线性地址空间容量为 4GB。分段地址转换的基本过程如图 2-2 所示。

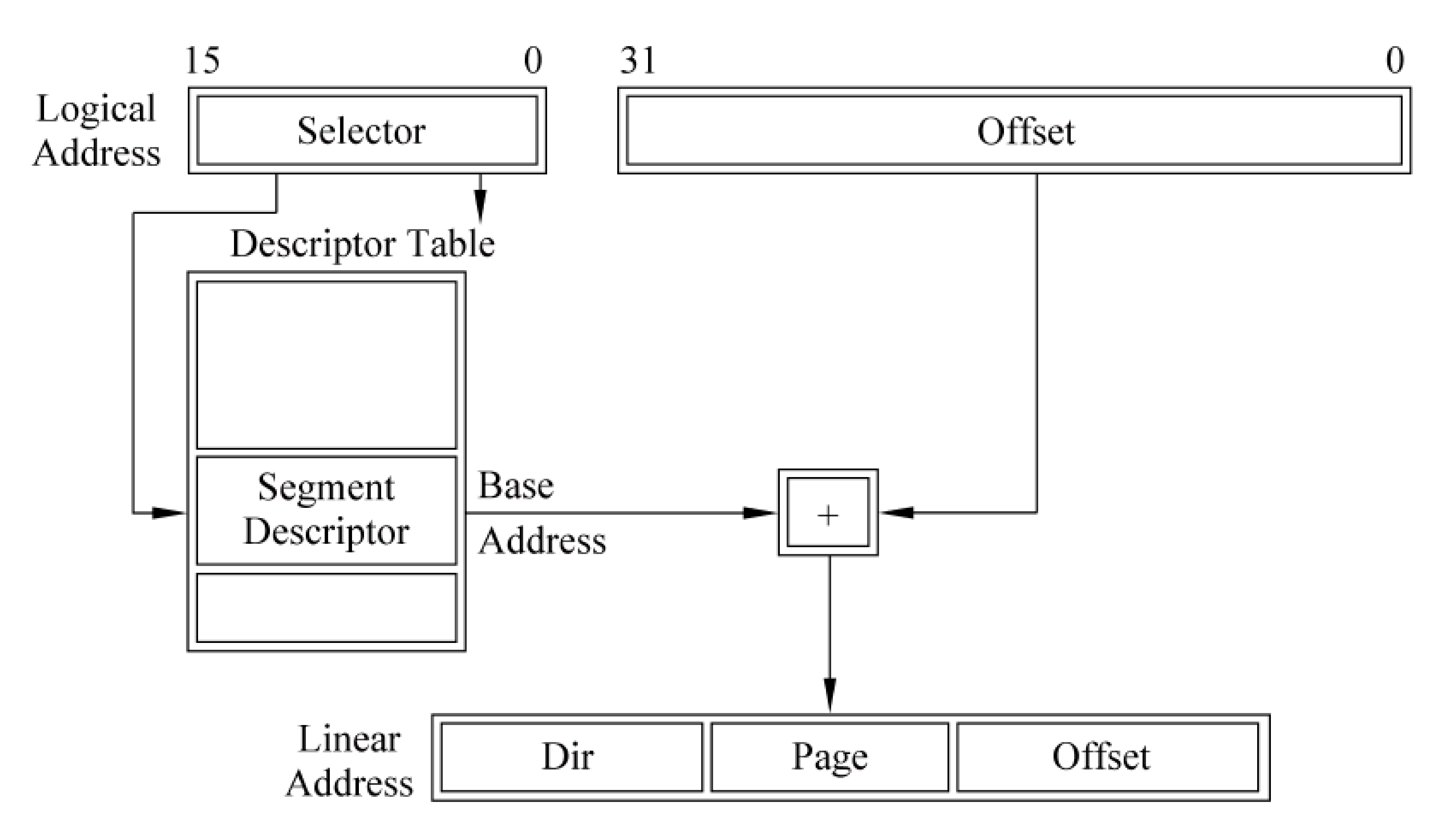


图 2-2　分段地址转换基本过程

分段存储管理机制需要在启动保护模式的前提下建立。从图 2-2 可以看出，为了使分段存储管理机制正常运行，需要建立好段描述符和段描述符表(参见 bootasm. S、mmu. h、pmm. c)。

(1) 段描述符。

在分段存储管理机制的保护模式下，每个段由段基地址(Base Address)、段界限(Limit)和段属性(Attributes)三个参数进行定义。在 ucore 中的 kern/mm/mmu. h 中的 struct segdesc 数据结构中有具体的定义。

① 段基地址：规定线性地址空间中段的起始地址。在 80386 保护模式下，段基地址长 32 位。因为基地址长度与寻址地址的长度相同，所以任何一个段都可以从 32 位线性地址空间中的任何一个字节开始，而不像实方式下规定的边界必须被 16 整除。

② 段界限：规定段的大小。在 80386 保护模式下，段界限用 20 位表示，而且段界限可以是以字节(B)为单位或以 4KB 为单位。

③ 段属性：确定段的各种性质。

a. 段属性中的粒度位(Granularity)，用符号 G 标记。G=0 表示以段界限为单位，20 位的界限可表示的范围是 1B～1MB，增量为 1B；G=1 表示段界限以 4KB 为单位，于是 20 位的界限可表示的范围是 4KB～4GB，增量为 4KB。

b. 类型(Type)：用于区别不同类型的描述符。可表示所描述的段是代码段还是数据段，所描述的段是否可读/写/执行，段的扩展方向等。

c. 描述符特权级(Descriptor Privilege Level，DPL)：用来实现保护机制。

d. 段存在位(Segment-Present bit)：如果这一位为 0，则此描述符为非法的，不能被用

来实现地址转换。如果一个非法描述符被加载进一个段寄存器，处理器会立即产生异常。图 2-3 显示了当存在位为 0 时，描述符的格式。操作系统可以任意地使用被标识为可用(Available)的位。

e. 已访问位(Accessed bit)：当处理器访问该段(当一个指向该段描述符的选择子被加载进一个段寄存器)时，将自动设置访问位。操作系统可清除该位。

上述参数通过段描述符来表示，段描述符的结构如图 2-3 所示。

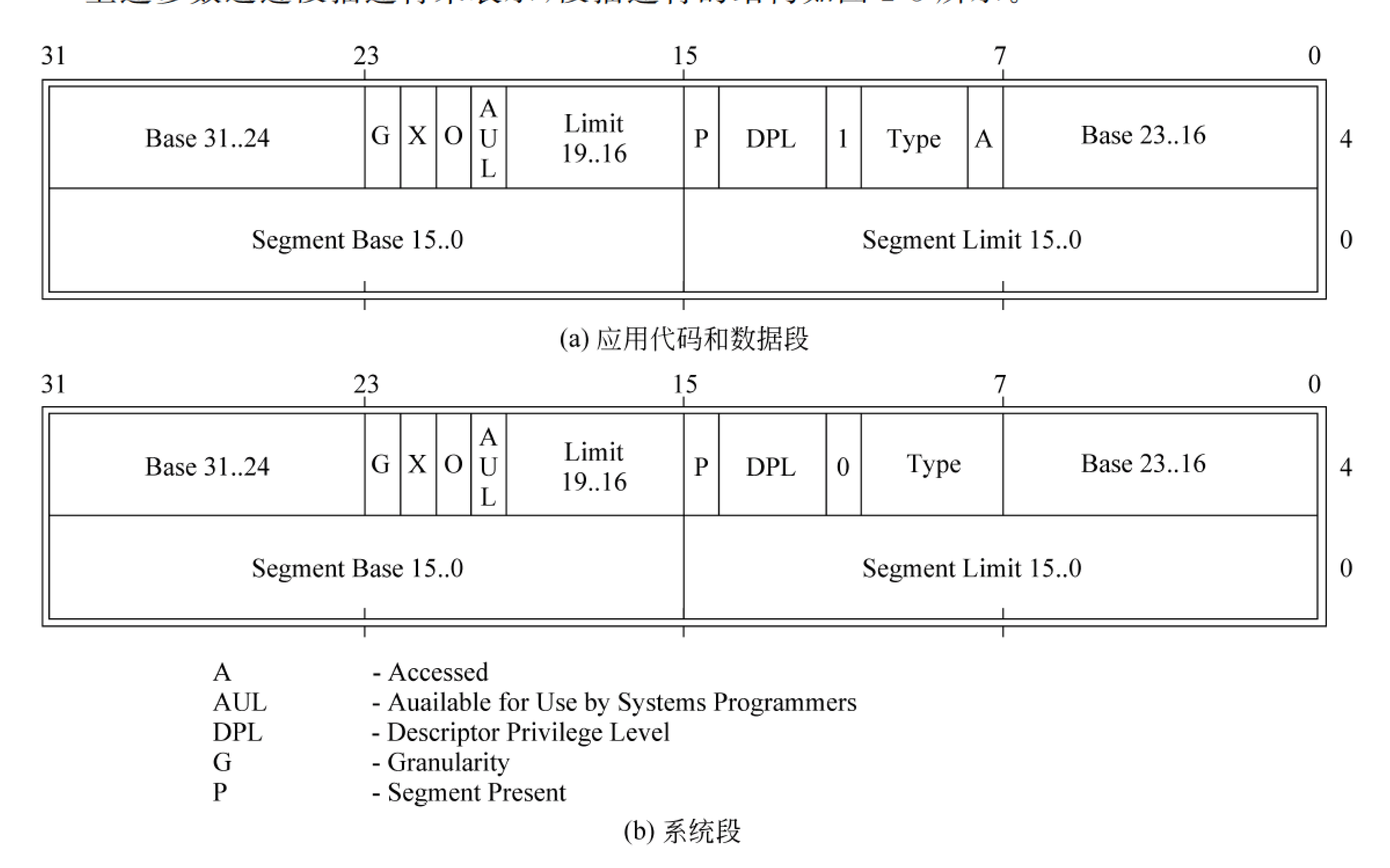


图 2-3　段描述符结构

(2) 全局描述符表。

全局描述符表是一个保存多个段描述符的“数组”，其起始地址保存在全局描述符表寄存器 GDTR 中。GDTR 长 48 位，其中高 32 位为基地址，低 16 位为段界限。由于 GDT 不能有 GDT 本身之内的描述符进行描述定义，所以处理器采用 GDTR 为 GDT 这一特殊的系统段。注意，全部描述符表中第一个段描述符设定为空段描述符。GDTR 中的段界限以字节为单位。对于含有 N 个描述符的描述符表的段界限通常可设为 $8\times N-1$。在 ucore 中的 boot/bootasm.S 中的 GDT 地址处和 kern/mm/pmm.c 中的全局变量数组 gdt[]分别有基于汇编语言和 C 语言的全局描述符表的具体实现。

(3) 选择子。

线性地址部分的**选择子**是用来选择哪个描述符表和在该表中索引一个描述符的。选择子可以作为指针变量的一部分，从而对应用程序员是可见的，但是一般是由连接加载器来设置。选择子的格式如图 2-4 所示。

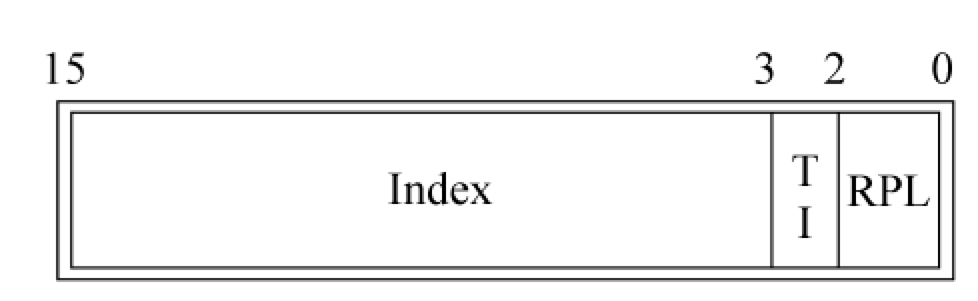


图 2-4　段选择子结构

① 索引(Index)：从描述符表中的 8192 个描

述符中选择一个描述符。处理器自动将这个索引值乘以 8(描述符的长度),再加上描述符表的基址来索引描述符表,从而选出一个合适的描述符。

② 表指示位(Table Indicator,TI):选择应该访问哪一个描述符表。0 代表应该访问全局描述符表(GDT),1 代表应该访问局部描述符表(LDT)。

③ 请求特权级(Requested Privilege Level,RPL):保护机制,在后续实验中会进一步讲解。

全局描述符表的第一项不能被 CPU 使用,所以当一个段选择子的索引(Index)部分和表指示位(Table Indicator)都为 0 时(即段选择子指向全局描述符表的第一项时),可以当做一个空的选择子(见 mmu.h 中的 SEG_NULL)。当一个段寄存器被加载一个空选择子时,处理器并不会产生一个异常。但是,当用一个空选择子去访问内存时,则会产生异常。

4) 保护模式下的特权级

在保护模式下,特权级总共有 4 个,编号从 0(最高特权)到 3(最低特权)。有 3 种主要的资源受到保护:内存、I/O 端口以及执行特殊机器指令的能力。在任一时刻,x86 CPU 都是在一个特定的特权级下运行的,从而决定了代码可以做什么,不可以做什么。这些特权级经常被称为保护环(Protection Ring),最内的环(ring 0)对应于最高特权 0,最外面的环(ring 3)一般给应用程序使用,对应最低特权 3。在 ucore 中,CPU 只用到其中的 2 个特权级:0(内核态)和 3(用户态)。

有大约 15 条机器指令被 CPU 限制只能在内核态执行,这些机器指令如果被用户模式的程序所使用,就会颠覆保护模式的保护机制并引起混乱,所以它们被保留给操作系统内核使用。如果试图在 ring 0 以外运行这些指令,就会导致一个一般保护异常(General-protection Exception)。对内存和 I/O 端口的访问也受类似的特权级限制。

数据段选择子的整个内容可由程序直接加载到各个段寄存器(如 SS 或 DS 等)中。这些内容里包含了请求特权级字段。然而,代码段寄存器(CS)的内容不能由装载指令(如 MOV)直接设置,而只能被那些会改变程序执行顺序的指令(如 JMP、INT、CALL)间接地设置,而且 CS 拥有一个由 CPU 维护的**当前特权级**字段(Current Privilege Level,CPL)。两者结构如图 2-5 所示。

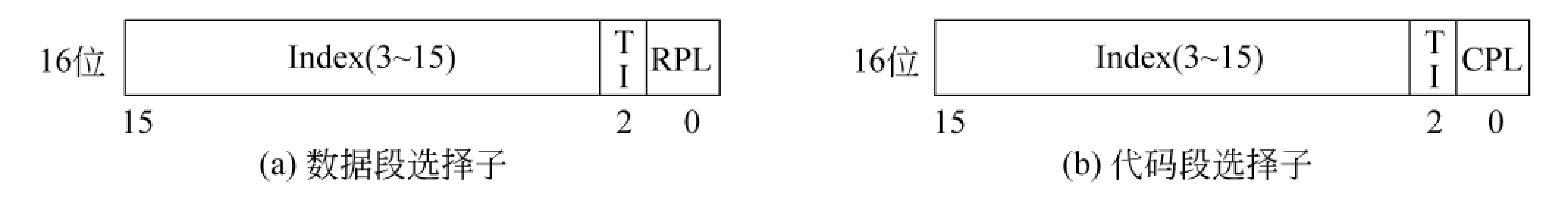


图 2-5　DS 和 CS 的结构图

代码段寄存器中的 CPL 字段(2 位)的值**总是等于** CPU 的当前特权级,所以只要看一眼 CS 中的 CPL,就可以知道此刻的特权级。

CPU 会在两个关键点上保护内存:当一个段选择符被加载时,以及当通过线性地址访问一个内存页时。因此,保护也反映在内存地址转换的过程之中,既包括分段又包括分页。当一个数据段选择符被加载时,就会发生如图 2-6 所示的检测过程。

因为数值越大特权越低,图 2-6 中的 MAX()用于选择 CPL 和 RPL 中特权最低的一个,并与描述符特权级(Descriptor Privilege Level,DPL)比较。如果 DPL 的值大于等于它,

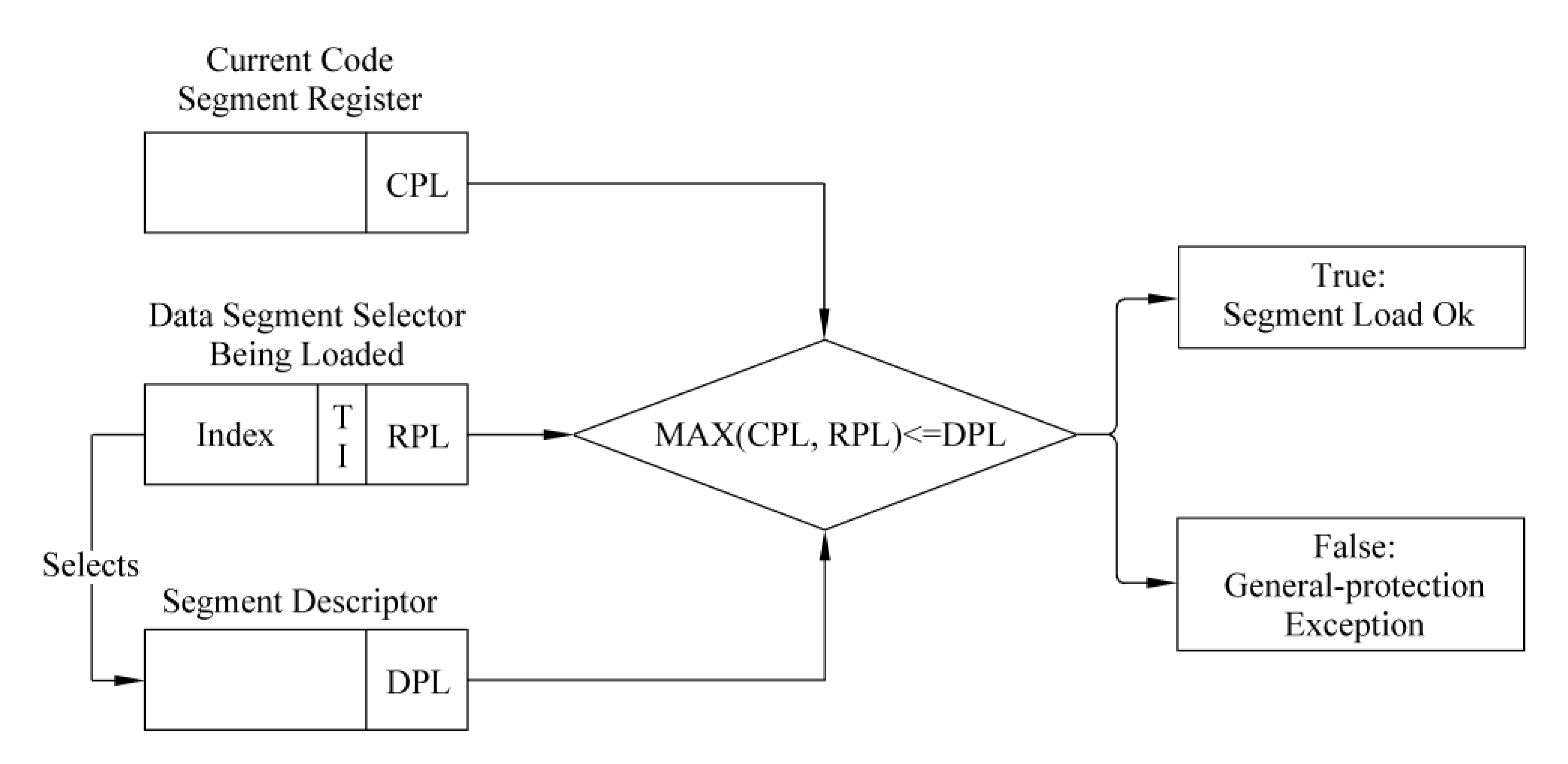


图 2-6　内存访问特权级检查过程

那么这个访问可正常进行了。RPL 背后的设计思想是：允许内核代码加载特权较低的段。例如，可以使用 RPL＝3 的段描述符来确保给定的操作所使用的段可以在用户模式中访问。但堆栈段寄存器是个例外，它要求 CPL、RPL 和 DPL 这 3 个值必须完全一致，才可以被加载。下面再总结一下 CPL、RPL 和 DPL。

（1）CPL：当前特权级(Current Privilege Level）保存在 CS 段寄存器（选择子）的最低两位，CPL 就是当前活动代码段的特权级，并且它定义了当前所执行程序的特权级别。

（2）DPL：描述符特权存储在段描述符中的权限位，用于描述对应段所属的特权等级，也就是段本身真正的特权级。

（3）RPL：请求特权级保存在选择子的最低两位。RPL 说明的是进程对段访问的请求权限，意思是当前进程想要的请求权限。RPL 的值由程序员自己设置，并不一定 RPL＞＝CPL，但是当 RPL＜CPL 时，实际起作用的就是 CPL 了，因为访问时的特权检查是判断 **max(RPL，CPL)＜＝DPL** 是否成立，所以 RPL 可以看成每次访问时的附加限制，RPL＝0 时附加限制最小，RPL＝3 时附加限制最大。

2．地址空间

分段机制涉及逻辑地址（Logical Address，应用程序员看到的地址，在操作系统原理上称为虚拟地址，以后提到虚拟地址就是指逻辑地址）、物理地址（Physical Address，实际的物理内存地址）、段描述符表（包含多个段描述符的数组）、段描述符（描述段的属性，及段描述符表这个数组中的数组元素）和段选择子（即段寄存器中的值，用于定位段描述符表中段描述符表项的索引）。

1）逻辑地址空间

从应用程序的角度看，逻辑地址空间就是应用程序员编程所用到的地址空间，例如，下面的程序片段：

```
int val=100;
int * point=&val;
```

其中指针变量 point 中存储的就是一个逻辑地址。在基于 80386 的计算机系统中，逻辑地址由一个 16 位的段寄存器（也称为段选择子）和一个 32 位的偏移量构成。

2）物理地址空间

从操作系统的角度看，CPU、内存硬件（通常说的内存条）和各种外设是它主要管理的硬件资源，而内存硬件和外设分布在物理地址空间中。物理地址空间就是一个“大数组”，CPU 通过索引（物理地址）来访问这个“大数组”中的内容。物理地址是指 CPU 提交到内存总线上用于访问计算机内存和外设的最终地址。

物理地址空间的大小取决于 CPU 实现的物理地址位数，在基于 80386 的计算机系统中，CPU 的物理地址空间为 4GB，如果计算机系统实际上有 1GB 物理内存（即通常说的内存条），而其他硬件设备的 I/O 寄存器映射到起始物理地址为 3GB 的 256MB 大小的地址空间，则该计算机系统的物理地址空间如图 2-7 所示。

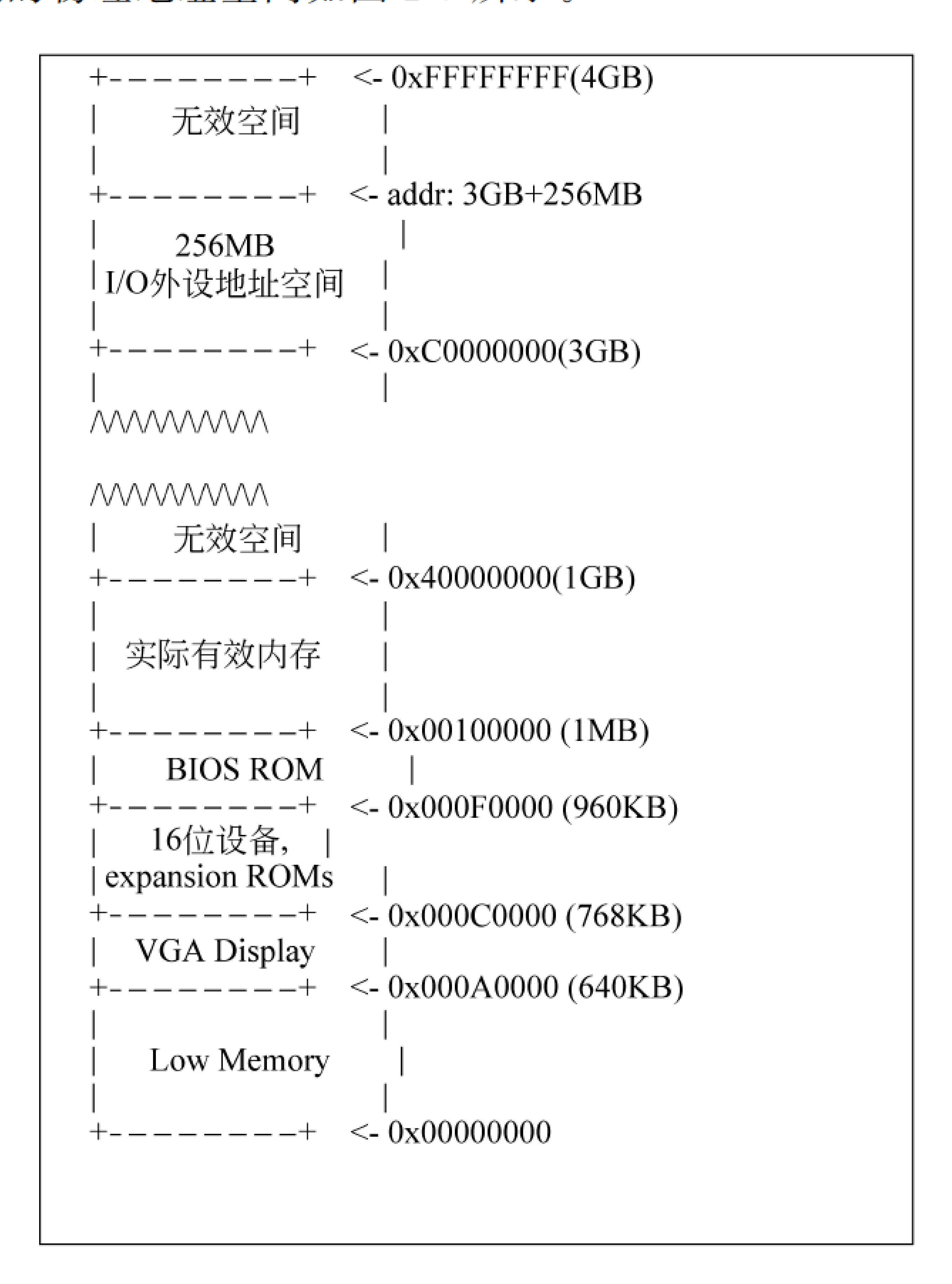


图 2-7　x86 计算机系统的物理地址空间

3）线性地址空间

一台计算机只有一个物理地址空间，但在操作系统的管理下，每个程序都认为自己独占整个计算机的物理地址空间。为了让多个程序能够有效地相互隔离和使用物理地址空间，引入线性地址空间（也称为虚拟地址空间）的概念。线性地址空间的大小取决于 CPU 实现的线性地址位数，在基于 80386 的计算机系统中，CPU 的线性地址空间为 4GB。线性地址空间会被映射到某一部分或整个物理地址空间，并通过索引（线性地址）来访问其中的内容。线性地址又称为虚拟地址，是进行逻辑地址转换后形成的地址索引，用于寻址线性地址空间。CPU 未启动分页机制时，线性地址等于物理地址；当 CPU 启动分页机制时，线性地址还需经过分页地址转换形成物理地址后，CPU 才能访问内存硬件和外设。三种地址的关系

如下所示。

① 启动分段机制，未启动分页机制：逻辑地址——>(分段地址转换)——>线性地址==物理地址。

② 启动分段和分页机制：逻辑地址——>(分段地址转换)——>线性地址——>分页地址转换)——>物理地址。

在操作系统的管理下，采用灵活的内存管理机制，在只有一个物理地址空间的情况下，可以存在多个线性地址空间。

3. 硬盘访问概述

bootloader 让 CPU 进入保护模式后，下一步的工作就是从硬盘上加载并运行 OS。考虑到实现的简单性，bootloader 的访问硬盘都是 LBA 模式的 PIO(Program I/O)方式，即所有的 I/O 操作是通过 CPU 访问硬盘的 I/O 地址寄存器完成。

一般主板有 2 个 IDE 通道，每个通道可以接 2 个 IDE 硬盘。访问第一个硬盘的扇区可设置 I/O 地址寄存器 0x1f0～0x1f7 实现的，具体参数如表 2-1 所示。一般第一个 IDE 通道通过访问 I/O 地址 0x1f0～0x1f7 来实现，第二个 IDE 通道通过访问 0x170～0x17f 实现。每个通道的主从盘的选择通过第 6 个 I/O 偏移地址寄存器来设置。磁盘 I/O 地址及对应功能如表 2-1 所示。

表 2-1　磁盘 I/O 地址及对应功能

I/O 地址	功　　能
0x1f0	读数据，当 0x1f7 不为忙状态时，可以读
0x1f1	可获取详细错误信息
0x1f2	要读写的扇区数，每次读写前，需要表明要读写几个扇区。最小是 1 个扇区
0x1f3	如果是 LBA 模式，就是 LBA 参数的 0～7 位
0x1f4	如果是 LBA 模式，就是 LBA 参数的 8～15 位
0x1f5	如果是 LBA 模式，就是 LBA 参数的 16～23 位
0x1f6	第 0～3 位：如果是 LBA 模式就是 24～27 位　第 4 位：为 0 主盘；为 1 从盘 第 6 位：为 1 是 LBA 模式；为 0 是 CHS 模式　第 5 位和第 7 位必须为 1
0x1f7	状态和命令寄存器。操作时先给命令，再读取，如果不是忙状态就从 0x1f0 端口读数据

当前硬盘数据储存到硬盘扇区中，一个扇区大小为 512B。读一个扇区的流程(可参看 boot/bootmain.c 中的 readsect 函数实现)大致如下。

(1) 等待磁盘准备好。

(2) 发出读取扇区的命令。

(3) 等待磁盘准备好。

(4) 把磁盘扇区数据读到指定内存。

4. ELF 文件格式概述

ELF(Executable and Linking Format)文件格式是 Linux 系统下的一种常用目标文件(Object File)格式，有三种主要类型。

(1) 用于执行的可执行文件(Executable File)，它用于提供程序的进程映像，加载的内

存执行。这也是本实验的 OS 文件类型。

(2) 用于链接的可重定位文件(Relocatable File),它可与其他目标文件一起创建可执行文件和共享目标文件。

(3) 共享目标文件(Shared Object File),链接器可将它与其他可重定位文件和共享目标文件链接成其他的目标文件,动态链接器又可将它与可执行文件和其他共享目标文件结合起来创建一个进程映像。

这里只分析与本实验相关的 ELF 可执行文件类型。ELF header 在文件开始处描述了整个文件的组织。ELF 的文件头包含整个执行文件的控制结构,其定义在 elf. h 中。

```
struct elfhdr {
    uint magic;                    //must equal ELF_MAGIC
    uchar elf[12];
    ushort type;
    ushort machine;
    uint version;
    uint entry;                    //程序入口的虚拟地址
    uint phoff;                    //program header 表的位置偏移
    uint shoff;
    uint flags;
    ushort ehsize;
    ushort phentsize;
    ushort phnum;                  //program header 表中的入口数目
    ushort shentsize;
    ushort shnum;
    ushort shstrndx;
};
```

program header 描述与程序执行直接相关的目标文件结构信息,用来在文件中定位各个段的映像,同时包含其他一些用来为程序创建进程映像所必需的信息。可执行文件的程序头部是一个 program header 结构的数组,每个结构描述了一个段或者系统准备程序执行所必需的其他信息。目标文件的“段”包含一个或者多个“节区”(Section),也就是“段内容(Segment Contents)”。程序头部仅对于可执行文件和共享目标文件有意义。可执行目标文件在 ELF 头部的 e_phentsize 和 e_phnum 成员中给出其自身程序头部的大小。程序头部的数据结构如下所示:

```
struct proghdr {
    uint type;                     //段类型
    uint offset;                   //段相对文件头的偏移值
    uint va;                       //段的第一个字节将被放到内存中的虚拟地址
    uint pa;
    uint filesz;
    uint memsz;                    //段在内存映像中占用的字节数
    uint flags;
    uint align;
};
```

根据 elfhdr 和 proghdr 的结构描述，bootloader 就可以完成对 ELF 格式的 ucore 操作系统的加载过程（参见 boot/bootmain.c 中的 bootmain 函数）。

补充材料

Link Address 是指编译器指定代码和数据所需要放置的内存地址，由链接器配置。Load Address 是指程序被实际加载到内存的位置（由程序加载器 ld 配置）。一般由可执行文件结构信息和加载器可保证这两个地址相同。Link Addr 和 LoadAddr 不同会导致如下问题。

（1）直接跳转位置错误。

（2）直接内存访问（只读数据区或 bss 等直接地址访问）错误。

（3）堆和栈等的使用不受影响，但是可能会覆盖程序、数据区域。

注意：*也存在 Link 地址和 Load 地址不一样的情况（例如，动态链接库）。*

2.3.3 操作系统启动过程

当 bootloader 通过读取硬盘扇区把 ucore 在系统加载到内存后，就转跳到 ucore 操作系统在内存中的入口位置（kern/init.c 中的 kern_init 函数的起始地址），这样 ucore 就接管了整个控制权。当前的 ucore 功能很简单，只完成基本的内存管理和外设中断管理。ucore 主要完成的工作如下。

（1）初始化终端。

（2）显示字符串。

（3）显示堆栈中的多层函数调用关系。

（4）切换到保护模式，启用分段机制。

（5）初始化中断控制器，设置中断描述符表，初始化时钟中断，使能整个系统的中断机制。

（6）执行 while（1）死循环。

以后的实验中会大量涉及各个函数直接的调用关系，以及由于中断处理导致的异步现象，可能对大家实现操作系统和改正其中的错误有很大影响。理解好函数调用关系的建立机制和中断处理机制，对后续实验会有很大帮助。下面就练习 5 涉及的函数栈调用关系和练习 6 中的中断机制的建立进行阐述。

1. 函数堆栈

栈是一个很重要的编程概念（编译课和程序设计课都讲过相关内容），与编译器和编程语言有紧密的联系。理解调用栈最重要的两点是栈的结构和 EBP 寄存器的作用。一个函数调用动作可分解为零到多个 PUSH 指令（用于参数入栈），一个 CALL 指令。CALL 指令内部其实还暗含了一个将返回地址（即 CALL 指令下一条指令的地址）压栈的动作（由硬件完成）。几乎所有本地编译器都会在每个函数体之前插入类似如下的汇编指令：

```
pushl   %ebp
movl   %esp , %ebp
```

这样在程序执行到一个函数的实际指令前，已经有以下数据顺序入栈：参数、返回地址、ebp 寄存器。由此得到类似如图 2-8 的栈结构（参数入栈顺序跟调用方式有关，这里以 C 语言默认的 CDECL

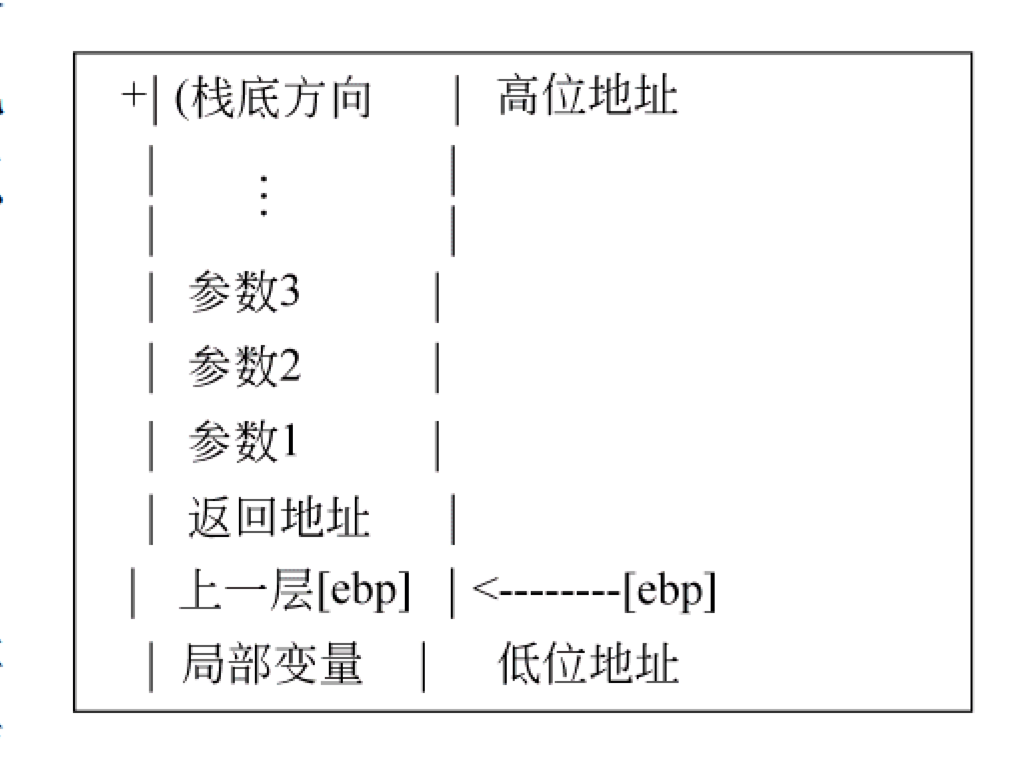


图 2-8 函数调用栈结构

为例）。

这两条汇编指令的含义是：首先将ebp寄存器入栈，然后将栈顶指针esp赋值给ebp。“mov ebp esp”这条指令表面上看是用esp覆盖ebp原来的值，其实不然。因为给ebp赋值之前，原ebp值已经被压栈（位于栈顶），而新的ebp又恰恰指向栈顶。此时ebp寄存器就已经处于一个非常重要的地位，该寄存器中存储着栈中的一个地址（原ebp入栈后的栈顶），从该地址为基准，向上（栈底方向）能获取返回地址、参数值，向下（栈顶方向）能获取函数局部变量值，而该地址处又存储着上一层函数调用时的ebp值。

一般而言，ss:[ebp+4]处为返回地址，ss:[ebp+8]处为第一个参数值（最后一个入栈的参数值，此处假设其占用4B内存），ss:[ebp-4]处为第一个局部变量，ss:[ebp]处为上一层ebp值。由于ebp中的地址处总是“上一层函数调用时的ebp值”，而在每一层函数调用中，都能通过当时的ebp值“向上（栈底方向）”能获取返回地址、参数值，“向下（栈顶方向）”能获取函数局部变量值。如此形成递归，直至到达栈底。这就是函数调用栈。

提示：练习5的正确实现取决于对这一节的正确理解和掌握。

2. 中断与异常

操作系统需要对计算机系统中的各种外设进行管理，这就需要CPU和外设能够相互通信才行。一般外设的速度远慢于CPU的速度。如果让操作系统通过CPU“主动关心”外设的事件，即采用通常的轮询（Polling）机制，则太浪费CPU资源。所以需要操作系统和CPU能够一起提供某种机制，让外设在需要操作系统处理外设相关事件的时候，能够“主动通知”操作系统，即打断操作系统和应用的正常执行，让操作系统完成外设的相关处理，然后再恢复操作系统和应用的正常执行。在操作系统中，这种机制称为中断机制。中断机制给操作系统提供了处理意外情况的能力，同时它也是实现进程/线程抢占式调度的一个重要基石。但中断的引入导致了对操作系统的理解更加困难。

在操作系统中，有三种特殊的中断事件。由CPU外部设备引起的外部事件如I/O中断、时钟中断、控制台中断等是异步产生的（即产生的时刻不确定），与CPU的执行无关，则称之为异步中断（Asynchronous Interrupt）也称外部中断，简称中断（Interrupt）。而把在CPU执行指令期间检测到不正常的或非法的条件（如除零错、地址访问越界）所引起的内部事件称为同步中断（Synchronous Interrupt），也称内部中断，简称异常（Exception）。把在程序中使用请求系统服务的系统调用而引发的事件，称作陷入中断（Trap Interrupt），也称软中断（Soft Interrupt），系统调用（System Call）简称Trap。在后续实验中会进一步讲解系统调用。

本实验只描述保护模式下的处理过程。当CPU收到中断（通过8259A完成，有关8259A的信息请看本章附录A）或者异常的事件时，它会暂停执行当前的程序或任务，通过一定的机制跳转到负责处理这个信号的相关处理例程中，在完成对这个事件的处理后再跳回到刚才被打断的程序或任务中。中断向量和中断服务例程的对应关系主要是由IDT（中断描述符表）负责。操作系统在IDT中设置好各种中断向量对应的中断描述符，并把IDT的起始地址保存在IDTR寄存器中。在产生中断后，CPU先根据IDTR找到IDT的起始地址，再根据中断向量号在IDT表中查询到对应中断服务例程的起始地址。

1）中断描述符表

中断描述符表（Interrupt Descriptor Table）把每个中断或异常编号和一个指向中断服

务例程的描述符联系起来。同 GDT 一样，IDT 是一个 8B 的描述符数组，但 IDT 的第一项可以包含一个描述符。CPU 把中断(异常)号乘以 8 作为 IDT 的索引。IDT 可以位于内存的任意位置，CPU 通过 IDT 寄存器(IDTR)的内容来寻址 IDT 的起始地址。指令 LIDT (Load IDT Register)和 SIDT(Store IDT Register)用来操作 IDTR。两条指令都有一个显示的操作数：一个 6B 表示的内存地址。指令的含义如下。

(1) LIDT 指令：使用一个包含线性地址基址和界限的内存操作数来加载 IDT。操作系统创建 IDT 时需要执行它来设定 IDT 的起始地址。这条指令只能在特权级 0 执行(可参见 libs/x86.h 中的 lidt 函数实现，其实就是一条汇编指令)。

(2) SIDT 指令：复制 IDTR 的基址和界限部分到一个内存地址。这条指令可以在任意特权级执行。

IDT 和 IDTR 寄存器的结构和关系如图 2-9 所示。

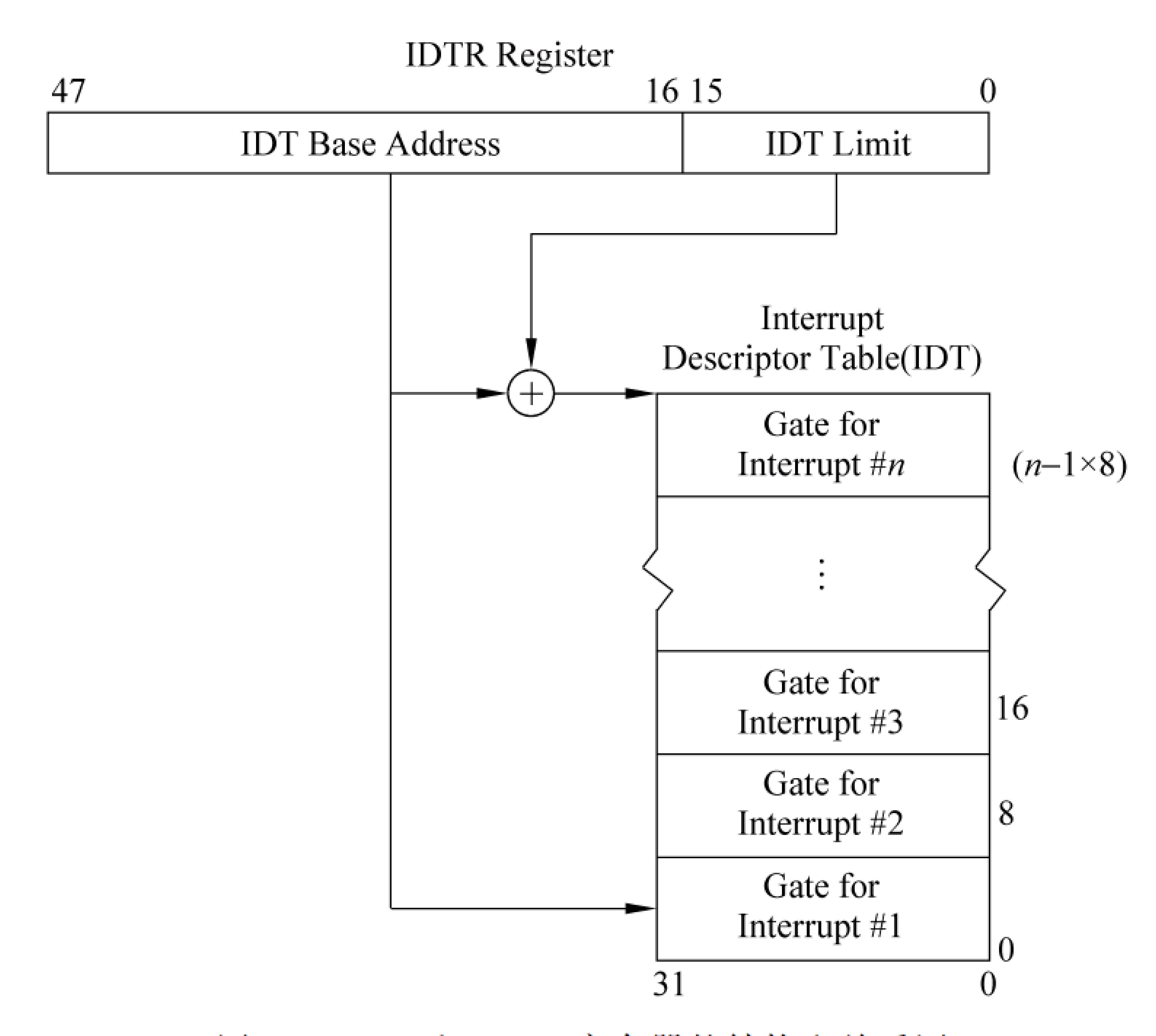


图 2-9　IDT 和 IDTR 寄存器的结构和关系图

在保护模式下，最多会存在 256 个 Interrupt/Exception Vectors。范围 0～31 内的 32 个向量被异常 Exception 和 NMI 使用，但当前并非所有这 32 个向量都已经被使用，有几个当前没有被使用的，请不要擅自使用它们，它们被保留，以备将来可能增加新的 Exception。范围 32～255 内的向量被保留给用户定义的 Interrupts。Intel 没有定义，也没有保留这些 Interrupts。用户可以将它们用作外部 I/O 设备中断(8259A IRQ)，或者系统调用(System Call、Software Interrupts)等。

2) IDT Gate Descriptors

Interrupts/Exceptions 应该使用 Interrupt Gate 和 Trap Gate，它们之间的唯一区别是：当调用 Interrupt Gate 时，Interrupt 会被 CPU 自动禁止；而调用 Trap Gate 时，CPU 则不会去禁止或打开中断，而是保留它原来的样子。在 IDT 中，可以包含如下 3 种类型的 Descriptor。

(1) TaskGate Descriptor(这里没有使用)。

(2) InterruptGate Descriptor(中断方式用到)。

(3) TrapGate Descriptor(系统调用用到)。

图 2-10 显示了 80386 的任务门描述符、中断门描述符、陷阱门描述符的格式。

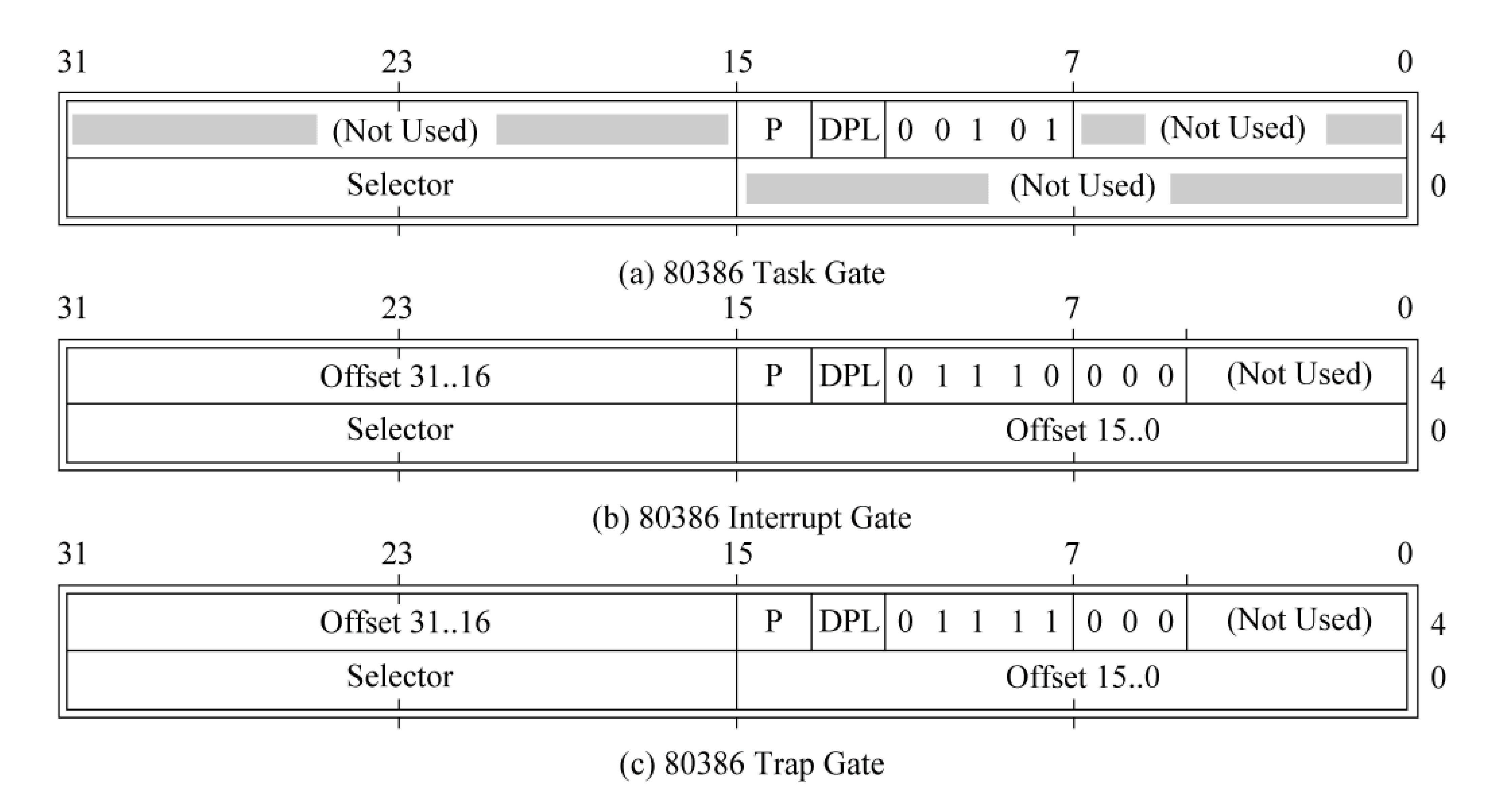


图 2-10　x86 的各种门的格式

可参见 kern/mm/mmu.h 中的 struct gatedesc 数据结构对中断描述符的具体定义。

3) 中断处理中硬件负责完成的工作

中断服务例程包括具体负责处理中断(异常)的代码,这些代码是操作系统的重要组成部分。需要注意区别的是,有两个硬件中断处理。

(1) 硬件中断处理过程 1(起始):从 CPU 收到中断事件后,打断当前程序或任务的执行,根据某种机制跳转到中断服务例程去执行。其具体流程如下。

① CPU 在执行完当前程序的每一条指令后,都会去确认在执行刚才的指令过程中中断控制器(如 8259A)是否发送中断请求过来,如果有那么 CPU 就会在相应的时钟脉冲到来时从总线上读取中断请求对应的中断向量。

② CPU 根据得到的中断向量(以此为索引)到 IDT 中找到该向量对应的中断描述符,中断描述符里保存着中断服务例程的段选择子。

③ CPU 使用 IDT 查到的中断服务例程的段选择子从 GDT 中取得相应的段描述符,段描述符中保存了中断服务例程的段基址和属性信息,此时 CPU 就得到了中断服务例程的起始地址,并跳转到该地址。

④ CPU 会根据 CPL 和中断服务例程的段描述符的 DPL 信息确认是否发生了特权级的转换。比如当前程序正运行在用户态,而中断程序运行在内核态,则意味着发生了特权级的转换。这时 CPU 会从当前程序的 TSS 信息(该信息在内存中的起始地址存在 TR 寄存器中)中取得该程序的内核栈地址,即包括内核态的 SS 和 ESP 的值,并立即将系统当前使用的栈切换成新的内核栈。这个栈就是即将运行的中断服务程序要使用的栈。紧接着就将

当前程序使用的用户态的 SS 和 ESP 压到新的内核栈中保存起来。

⑤ CPU 需要开始保存当前被打断的程序的现场(即一些寄存器的值),以便将来恢复被打断的程序继续执行。这需要利用内核栈来保存相关现场信息,即依次压入当前被打断程序使用的 Eflags、CS、EIP、Error、Code(如果是有错误码的异常)信息。

⑥ CPU 利用中断服务例程的段描述符将其第一条指令的地址加载到 CS 和 EIP 寄存器中,开始执行中断服务例程。这意味着先前的程序被暂停执行,中断服务程序正式开始工作。

(2) 硬件中断处理过程 2(结束):每个中断服务例程在有中断处理工作完成后需要通过 IRET(或 IRETD)指令恢复被打断的程序的执行。CPU 执行 IRET 指令的具体过程如下。

① 程序执行这条 IRET 指令时,首先会从内核栈里弹出先前保存的被打断的程序的现场信息,即 Eflags、CS、EIP 重新开始执行。

② 如果存在特权级转换(从内核态转换到用户态),则还需要从内核栈中弹出用户态栈的 SS 和 ESP,这样也意味着栈也被切换回原先使用的用户态的栈了。

③ 如果此次处理的是带有错误码(Error Code)的异常,CPU 在恢复先前程序的现场时,并不会弹出 Error Code。这一步需要通过软件完成,即要求相关的中断服务例程在调用 IRET 返回之前添加出栈代码主动弹出 Error Code。

图 2-11 显示了从中断向量到 GDT 中相应中断服务程序起始位置的定位方式。

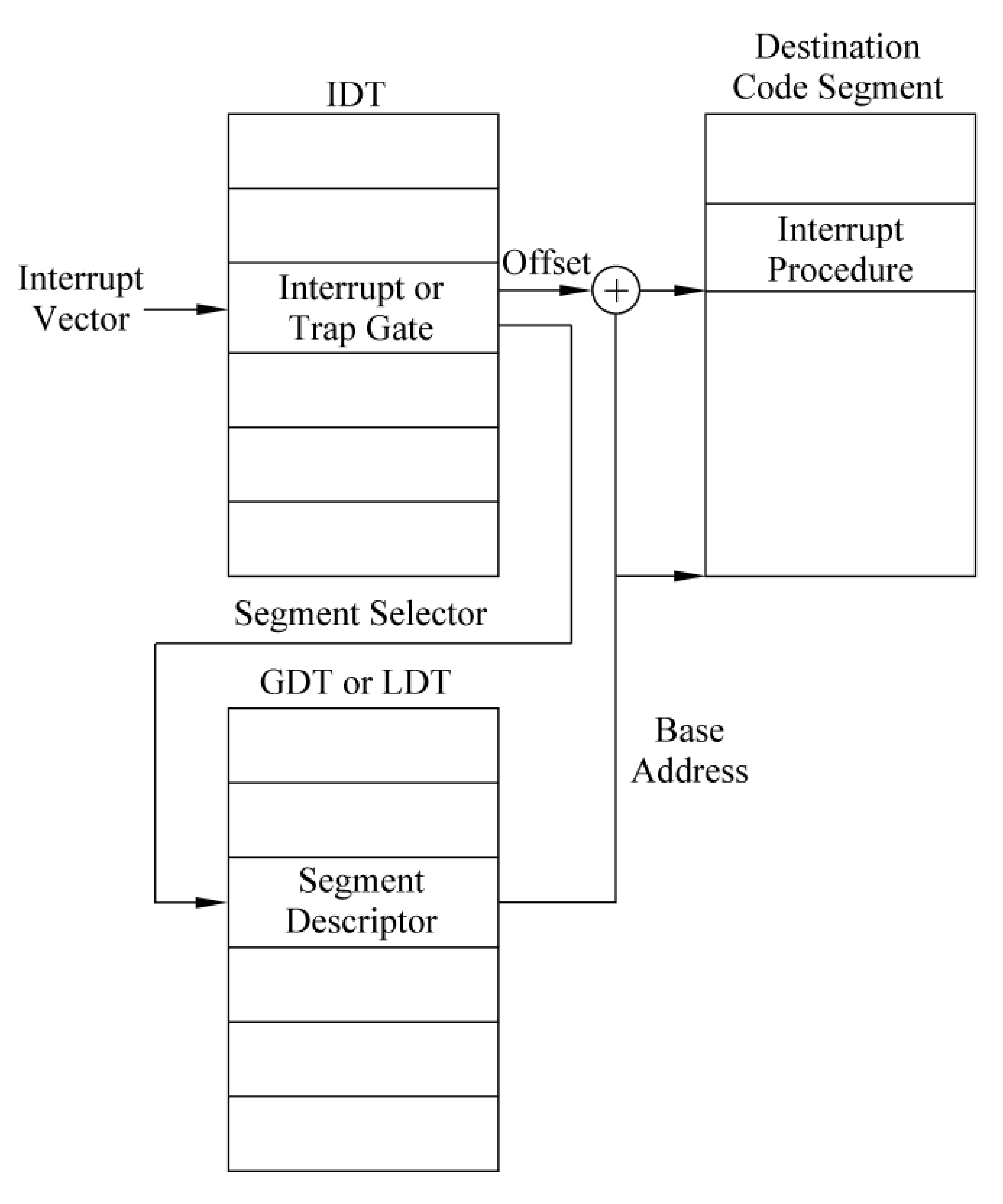
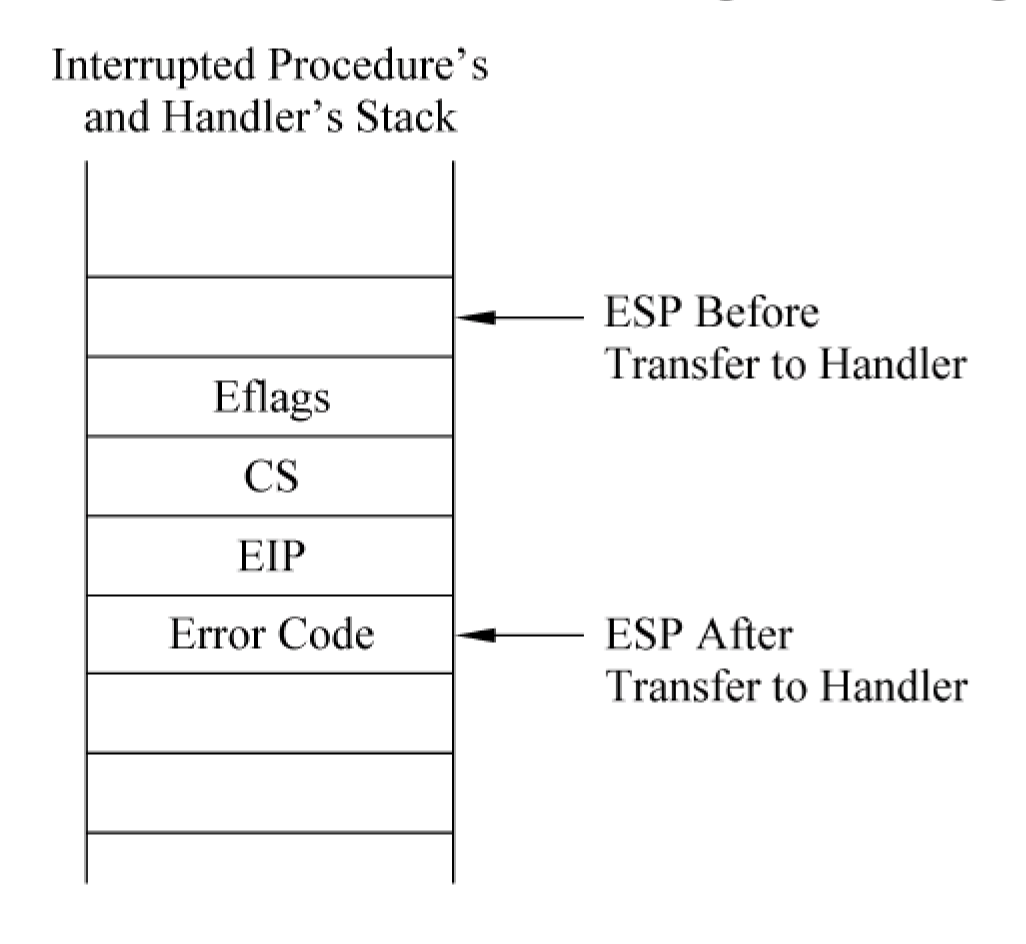


图 2-11　中断向量与中断服务例程起始地址的关系

4）中断产生后堆栈的变化

图 2-12 显示了给出相同特权级和不同特权级情况下中断产生后的堆栈变化示意图。



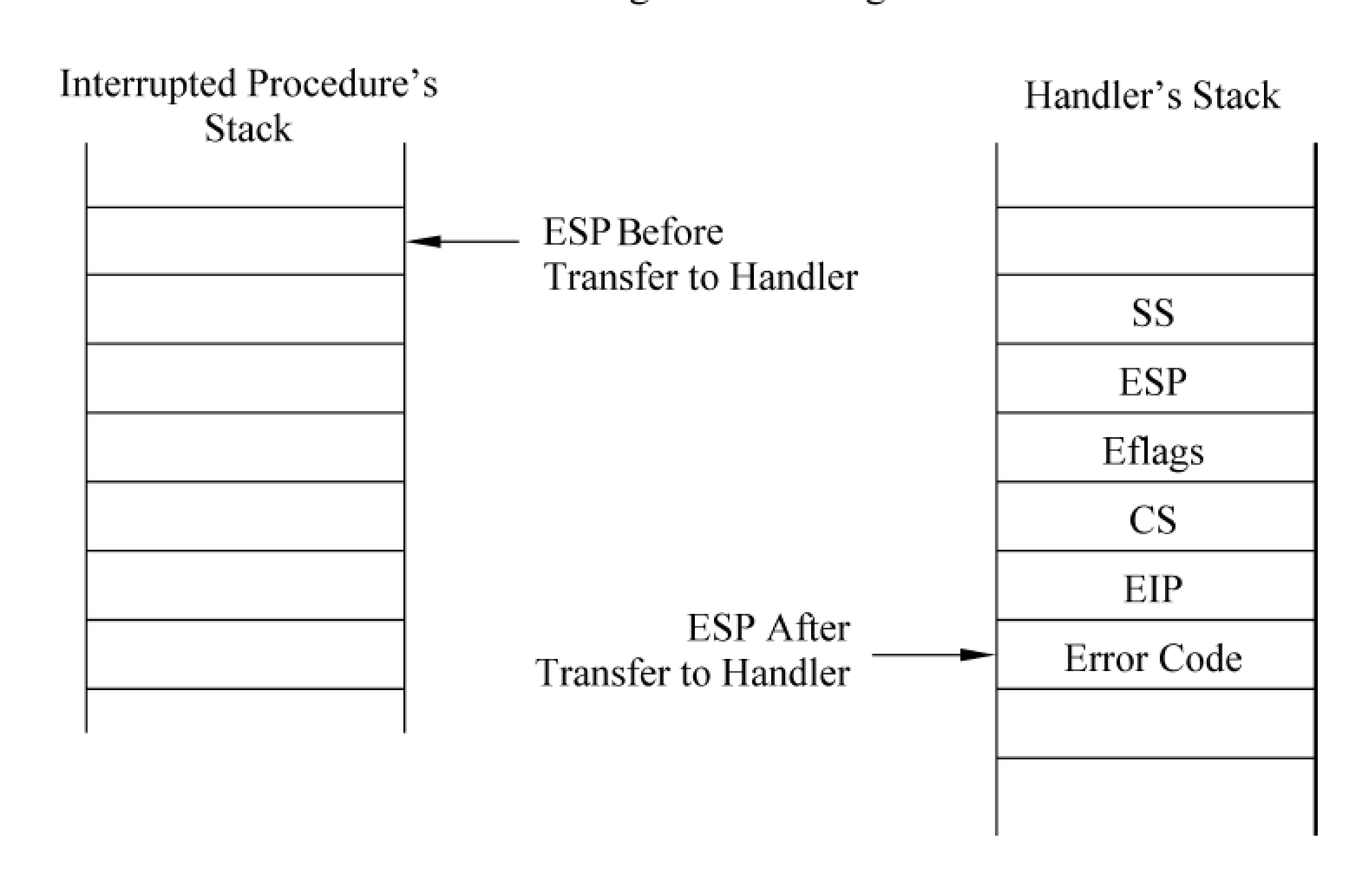


图 2-12　相同特权级和不同特权级情况下中断产生后的堆栈变化示意图

5）中断处理的特权级转换

中断处理的特权级转换是通过**门描述符**(Gate Descriptor)和**相关**指令来完成的。一个门描述符就是一个系统类型的段描述符，一共有 4 个子类型：调用门描述符(Call-gate Descriptor)、中断门描述符(Interrupt-gate Descriptor)、陷阱门描述符(Trap-gate Descriptor)和任务门描述符(Task-gate Descriptor)。与中断处理相关的是中断门描述符和陷阱门描述符。这些门描述符被存储在中断描述符表(Interrupt Descriptor Table，IDT)中。CPU 把中断向量作为 IDT 表项的索引，用来指出当中断发生时使用哪一个门描述符来处理中断。中断门描述符和陷阱门描述符几乎是一样的。中断发生时实施特权检查的过程如图 2-13 所示。

门中的 DPL 和段选择符一起控制着访问，同时，段选择符结合偏移量(Offset)指出了

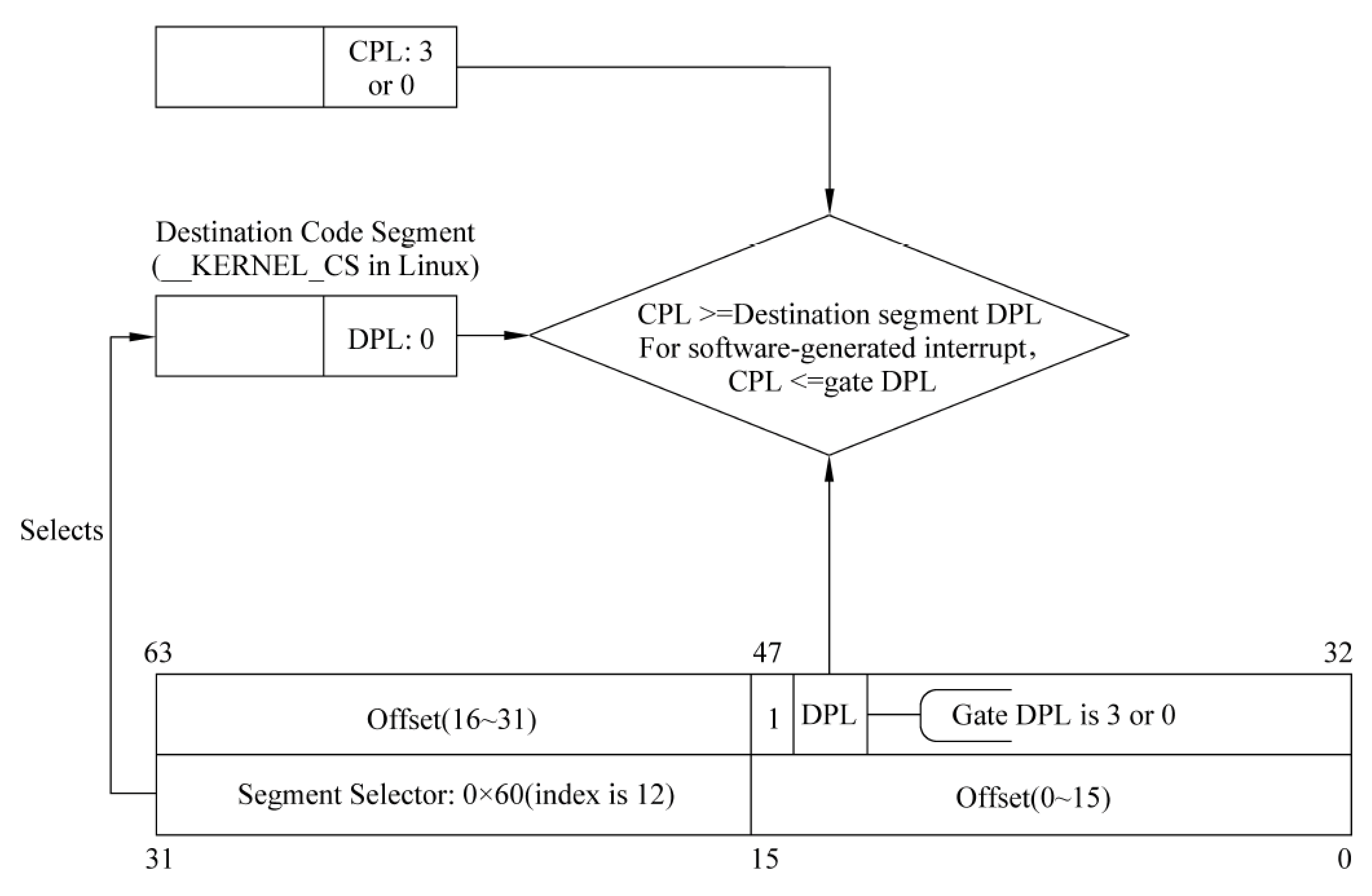


图 2-13　中断发生时实施特权检查的过程

中断处理例程的入口点。内核一般在门描述符中填入内核代码段的段选择子。产生中断后，CPU 一定不会将运行控制从高特权环转向低特权环，特权级必须要么保持不变（当操作系统内核自己被中断的时候），或被提升（当用户态程序被中断的时候）。无论哪一种情况，作为结果的 CPL 必须等于目的代码段的 DPL。如果 CPL 发生了改变，一个堆栈切换操作（通过 TSS 完成）就会发生。如果中断是被用户态程序中的指令所触发的（比如软件执行 INT n 生产的中断），还会增加一个额外的检查：门的 DPL 必须具有与 CPL 相同或更低的特权。这样就防止了用户代码随意触发中断。如果这些检查失败，会产生一个一般保护异常（General-protection Exception）。

3. lab1 中对中断的处理实现

1）外设基本初始化设置

Lab1 中实现了中断初始化和对键盘、串口、时钟外设进行中断处理。串口的初始化函数 serial_init（位于/kern/driver/console.c）中涉及中断初始化工作很简单：

```
⋮
//使能串口 1接收字符后产生中断
    outb(COM1+COM_IER, COM_IER_RDI);
⋮
//通过中断控制器使能串口 1中断
    pic_enable(IRQ_COM1);
```

键盘的初始化函数 kbd_init（位于 kern/driver/console.c 中）完成了对键盘的中断初始

化工作，具体操作更加简单：

```
    ⋮
//通过中断控制器使能键盘输入中断
pic_enable(IRQ_KBD);
```

时钟是一种有着特殊作用的外设，其作用并不仅仅是计时。在后续章节中将讲到，正是由于有了规律的时钟中断，才使得无论当前 CPU 运行在哪里，操作系统都可以在预先确定的时间点上获得 CPU 的控制权。这样当一个应用程序运行了一定时间后，操作系统会通过时钟中断获得 CPU 的控制权，并可把 CPU 资源让给更需要 CPU 的其他应用程序。时钟的初始化函数 clock_init（位于 kern/driver/clock.c 中）完成了对时钟控制器 8253 的初始化：

```
        ⋮
//设置时钟每秒中断 100 次
    outb(IO_TIMER1, TIMER_DIV(100) % 256);
    outb(IO_TIMER1, TIMER_DIV(100) / 256);
//通过中断控制器使能时钟中断
    pic_enable(IRQ_TIMER);
```

2）中断初始化设置

操作系统如果要正确处理各种不同的中断事件，就需要安排应该由哪个中断服务例程负责处理特定的中断事件。系统将所有的中断事件统一进行编号（0～255），这个编号称为中断向量。以 ucore 为例，操作系统内核启动以后，会通过 idt_init 函数初始化 idt 表（参见 trap.c），而其中 vectors 中存储了中断处理程序的入口地址。vectors 定义在 vector.S 文件中，通过一个工具程序 vector.c 生成。其中仅有 System call 中断的权限为用户权限（DPL_USER），即仅能够使用 int 0x30 指令。此外还有对 tickslock 的初始化，该锁用于处理时钟中断。

vector.S 文件通过 vectors.c 自动生成，其中定义了每个中断的入口程序和入口地址（保存在 vectors 数组中）。其中，中断可以分成两类：一类是压入错误编码的（Error Code），另一类不压入错误编码。对于第二类，vector.S 自动压入一个 0。此外，还会压入相应中断的中断号。在压入两个必要的参数之后，中断处理函数跳转到统一的入口 alltraps 处。

3）中断的处理过程

trap 函数（定义在 trap.c 中）是对中断进行处理的过程，所有的中断在经过中断入口函数__alltraps 预处理后（定义在 trapasm.S 中），都会跳转到这里。在处理过程中，根据不同的中断类型，进行相应的处理。在相应的处理过程结束以后，trap 将会返回，被中断的程序会继续运行。整个中断处理流程大致如图 2-14 所示。

至此，对整个 lab1 中的主要部分的背景知识和实现进行了阐述。请大家能够根据前面的练习要求完成所有的练习。

trapasm.S

(1) 产生中断后，CPU 跳转到相应的中断处理入口(Vectors)，并在栈中压入相应的 error_code(是否存在与异常号相关)以及 trap_no，然后跳转到 alltraps 函数入口。

注意：此处的跳转是 jump 过程

(high)	…
产生中断时的 eip→	eip
	error_code
esp→	trap_no
(low)	…

在栈中保存当前被打断程序的 trapframe 结构(参见过程 trapasm.S)。设置 kernel(内核)的数据段寄存器，最后压入 esp，作为 trap 函数参数(structtrapframe * tf)并跳转到中断处理函数 trap 处：

注意：此时的跳转是 call 调用，会压入返回地址 eip，注意区分此处 eip 与 trapframe 中的 eip：

trapframe 的结构为

Struct trapframe
{
uint edi;
uint esi;
uint ebp;
⋮
ushort es;
ushort padding1;
ushort ds;
ushort padding2;
uint trapno;
uint err;
uint eip;
⋮
}
	观察 trapframe 结构与中断产生过程的压栈顺序。
	需要明确 pushal 指令都保存了哪些寄存器，按照什么顺序？
	
	← trap_no
	← trap_error
	← 产生中断处的 eip

进入 trap 函数，对中断进行相应的处理：

(3) 结束 trap 函数的执行后，通过 ret 指令返回到 alltraps 执行过程。

从栈中恢复所有寄存器的值。

调整 esp 的值：跳过栈中的 trap_no 与 error_code，使 esp 指向中断返回 eip，通过 iret 调用恢复 cs、eflag 以及 eip，继续执行。

trap.c

(2) 详细的中断分类以及处理流程如下：

根据中断号对不同的中断进行处理。其中，若中断号是 IRQ_OFFSET+IRQ_TIMER，则为时钟中断，则 ticks 将增加 1。

若中断号是 IRQ_OFFSET+IRQ_COM1，则为串口中断，则显示收到的字符。

若中断号是 IRQ_OFFSET+IRQ_KBD，则为键盘中断，则显示收到的字符。

若为其他中断且产生在内核状态，则挂起系统。

图 2-14　ucore 中断处理流程

2.4 实验报告要求

从网站上下载 lab1. tar. bz2 后，解压得到代码目录 lab1，完成实验。在实践中完成实验的各个练习。在报告中回答所有练习中提出的问题。实验报告文档命名为 lab1-学生 ID. odt 或 lab1-学生 ID. txt。推荐用 txt 格式，即基本文本格式。对于 lab1 中编程任务，完成编写之后，在 lab1 目录下执行 make handin 任务，即会自动生成 lab1-handin. tar. gz 文件。建立一个目录，名字为 lab1_result，将实验报告文档和之前生成的 handin 文件放在该目录下。然后用 tar 软件压缩打包此目录，并命名为 lab1-学生 ID. tar. bz2（在 lab1_result 的上层目录下执行"tar jcf lab1-学生 ID. tar. bz2 lab1_result "即可）。最后请一定提前或按时提交到网络学堂上。

注意要有 lab1 的注释，代码中所有需要完成的地方（challenge 除外）都有 lab1 和"Your Code"的注释，请在提交时特别注意保持注释，并将"Your Code"替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

辅助材料 A 关于 A20 Gate

参考"关于 A20 Gate" http://hengch. blog. 163. com/blog/static/107800672009013104623747/和激活 A20 地址线详解"（http://wenku. baidu. com/view/d6efe68fcc22bcd126ff0c00. html）。

Intel 早期的 8086 CPU 提供了 20 根地址线，可寻址空间范围即 0～2^{20}（00000H～FFFFFH）的 1MB 内存空间。但 8086 的数据处理位宽为 16 位，无法直接寻址 1MB 内存空间，所以 8086 提供了段地址加偏移地址的地址转换机制。PC 的寻址结构是 Segment：Offset，Segment 和 Offset 都是 16 位的寄存器，最大值是 0ffffh，换算成物理地址的计算方法是把 Segment 左移 4 位，再加上 Offset，所以 Segment：Offset 所能表达的寻址空间最大应为 0ffff0h＋0ffffh＝10ffefh（前面的 0ffffh 是 Segment＝0ffffh 并向左移动 4 位的结果，后面的 0ffffh 是可能的最大 Offset），这个计算出的 10ffefh 是多大呢？大约是 1088KB，就是说，Segment：Offset 的地址表示能力，超过了 20 位地址线的物理寻址能力。所以当寻址到超过 1MB 的内存时，会发生"回卷"（不会发生异常）。但下一代的基于 Intel 80286 CPU 的 PC AT 计算机系统提供了 24 根地址线，这样 CPU 的寻址范围变为 2^{24}＝16MB，同时也提供了保护模式，可以访问到 1MB 以上的内存了，此时如果遇到"寻址超过 1MB"的情况，系统不会再"回卷"了，这就造成了向下不兼容。为了保持完全的向下兼容性，IBM 决定在 PC AT计算机系统上加个硬件逻辑，来模仿以上的回绕特征，于是出现了 A20 Gate。他们的方法就是把 A20 地址线控制和键盘控制器的一个输出进行 AND 操作，这样来控制 A20 地址线的打开（使能）和关闭（屏蔽/禁止）。一开始时 A20 地址线控制是被屏蔽的（总为 0），直到系统软件通过一定的 I/O 操作去打开它（参看 bootasm. S）。很显然，在实模式下要访问高端内存区，这个开关必须打开，在保护模式下，由于使用 32 位地址线，如果 A20 恒等于 0，那么系统只能访问奇数兆的内存，即只能访问 0～1M、2～3M、4～5M…，这显然是不行的，所以在保护模式下，这个开关也必须打开。

当 A20 地址线控制禁止时，则程序就像在 8086 中运行，1MB 以上的地址是不可访问

的。在保护模式下 A20 地址线控制是要打开的。为了使能所有地址位的寻址能力,必须向键盘控制器 8042 发送一个命令。键盘控制器 8042 将会将它的某个输出引脚的输出置高电平,作为 A20 地址线控制的输入。一旦设置成功之后,内存将不会再被绕回(Memory Wrapping),这样就可以寻址整个 286 的 16MB 内存,或者是寻址 80386 级别机器的所有 4G 内存了。

键盘控制器 8042 的逻辑结构图如图 2-15 所示。从软件的角度来看,如何控制 8042 呢?早期的 PC,控制键盘有一个单独的单片机 8042,现如今这个芯片已经给集成到了其他大片子中,但其功能和使用方法还是一样,当 PC 刚刚出现 A20 Gate 的时候,估计为节省硬件设计成本,工程师使用这个 8042 键盘控制器来控制 A20 Gate,但 A20 Gate 与键盘管理没有一点关系。下面先从软件的角度简单介绍一下 8042 这个芯片。

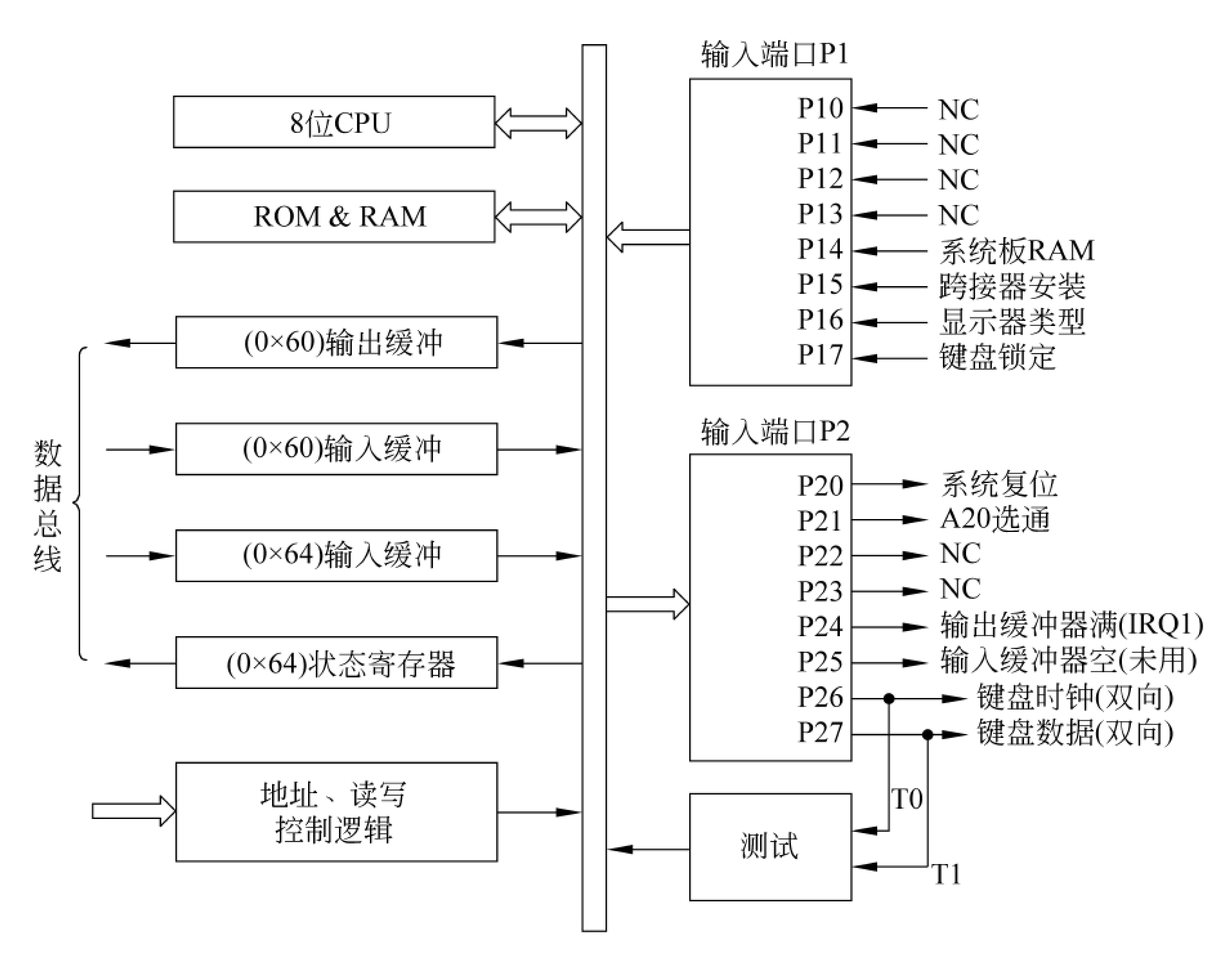


图 2-15　键盘控制器 8042 的逻辑结构图

8042 键盘控制器的 I/O 端口是 0x60～0x6f,实际上,IBM PC/AT 使用的只有 0x60 和 0x64 两个端口(0x61、0x62 和 0x63 用于与 XT 兼容)。8042 通过这些端口给键盘控制器或键盘发送命令或读取状态。输出端口 P2 用于特定目的。位 0(P20 引脚)用于实现 CPU 复位操作,位 1(P21 引脚)用户控制 A20 信号线的开启与否。系统向输入缓冲(端口 0x64)写入一个字节,即发送一个键盘控制器命令,可以带一个参数。参数是通过 0x60 端口发送的。命令的返回值也从端口 0x60 去读。8042 有 4 个寄存器。

(1) 1 个 8 位长的 Input Buffer:Write-Only。

(2) 1 个 8 位长的 Output Buffer:Read-Only。

(3) 1 个 8 位长的 Status Register:Read-Only。

(4) 1 个 8 位长的 Control Register:Read/Write。

有两个端口地址:60h 和 64h,有关对它们的读写操作描述如下。

(1) 读 60h 端口,读 Output Buffer。

(2) 写 60h 端口,写 Input Buffer。

(3) 读 64h 端口,读 Status Register。

(4) 操作 Control Register,首先要向 64h 端口写一个命令(20h 为读命令,60h 为写命令),然后根据命令从 60h 端口读出 Control Register 的数据或者向 60h 端口写入 Control Register 的数据(64h 端口还可以接受许多其他的命令)。

Status Register 的定义(要用 bit 0 和 bit 1)如下:

```
bit meaning
-----------------------------------------------------------------
0  Output Register(60h)中有数据
1  Input Register(60h/64h)有数据
2  系统标志(上电复位后被置为 0)
3  data in input register is command (1) or data (0)
4  1=keyboard enabled, 0=keyboard disabled (via switch)
5  1=transmit timeout (data transmit not complete)
6  1=receive timeout (data transmit not complete)
7  1=even parity rec'd, 0=odd parity rec'd (should be odd)
```

除了这些资源外,8042 还有 3 个内部端口:Input Port、Outport Port 和 Test Port,这三个端口的操作都是通过向 64h 发送命令,然后在 60h 进行读写的方式完成,其中本文要操作的 A20 Gate 被定义在 Output Port 的 bit 1 上,所以有必要对 Outport Port 的操作及端口定义做一个说明。

(1) 读 Output Port:向 64h 发送 0d0h 命令,然后从 60h 读取 Output Port 的内容。

(2) 写 Output Port:向 64h 发送 0d1h 命令,然后向 60h 写入 Output Port 的数据。

(3) 禁止键盘操作命令:向 64h 发送 0adh。

(4) 打开键盘操作命令:向 64h 发送 0aeh。

有了这些命令和知识,就可以实现操作 A20 Gate 从实模式切换到保护模式了。

理论上讲,我们只要操作 8042 芯片的输出端口(64h)的 bit 1,就可以控制 A20 Gate,但实际上,当准备向 8042 的输入缓冲区里写数据时,可能里面还有其他数据没有处理,所以,我们要首先禁止键盘操作,同时等待数据缓冲区中没有数据以后,才能真正地去操作 8042 打开或者关闭 A20 Gate。打开 A20 Gate 的具体步骤大致如下(参考 bootasm.S)。

(1) 等待 8042 Input Buffer 为空。

(2) 发送 Write 8042 Output Port (P2)命令到 8042 Input Buffer。

(3) 等待 8042 Input Buffer 为空。

(4) 将 8042 Output Port(P2)得到字节的第 2 位置 1,然后写入 8042 Input Buffer。

辅助材料 B　启动后第一条执行的指令

参考 IA-32 Intel Architecture Software Developer's Manual Volume 3:System Programming Guide Section 9.1.4。

9.1.4 First Instruction Executed

The first instruction that is fetched and executed following a hardware reset is located at physical address FFFFFFF0H. This address is 16 bytes below the processor's uppermost physical address. The EPROM containing the softwareinitialization code must be located at this address.

The address FFFFFFF0H is beyond the 1-MByte addressable range of the processor while in realaddress mode. The processor is initialized to this starting address as follows. The CS register has two parts: the visible segment selector part and the hidden base address part. In realaddress mode, the base address is normally formed by shifting the 16-bit segment selector value 4 bits to the left to produce a 20-bit base address. However, during a hardware reset, the segment selector in the CS register is loaded with F000H and the base address is loaded with FFFF0000H. The starting address is thus formed by adding the base address to the value in the EIP register (that is, FFFF0000+FFF0H=FFFFFFF0H).

The first time the CS register is loaded with a new value after a hardware reset, the processor will follow the normal rule for address translation in real-address mode (that is, [CS base address = CS segment selector * 16]). To insure that the base address in the CS register remains unchanged until the EPROM based softwareinitialization code is completed, the code must not contain a far jump or far call or allow an interrupt to occur (which would cause the CS selector value to be changed).

第 3 章　实验 2：物理内存管理

3.1　实 验 目 的

（1）理解基于段页式内存地址的转换机制。

（2）理解页表的建立和使用方法。

（3）理解物理内存的管理方法。

3.2　实 验 内 容

实验 2 过后大家做出来了一个可以启动的系统，实验 2 主要涉及操作系统的物理内存管理。操作系统为了使用内存，还需高效地管理内存资源。在实验 2 中大家会了解并且自己动手完成一个简单的物理内存管理系统。

本次实验包含三个部分。首先了解如何发现系统中的物理内存；然后了解如何建立对物理内存的初步管理，即了解连续物理内存管理；最后了解页表相关的操作，即如何建立页表来实现虚拟内存到物理内存之间的映射，对段页式内存管理机制有一个比较全面的了解。本实验里面实现的内存管理非常基本，并没有涉及对实际机器的优化，比如，针对 Cache 的优化等。实际操作系统（如 Linux 等）中的内存管理是相当复杂的。如果大家有兴趣，尝试完成扩展练习。

3.2.1　练习

练习 0：填写已有实验。

本实验依赖实验 1。请把要做的实验 1 的代码填入本实验中代码有 lab1 的注释相应部分。

提示：可采用 merge 工具，比如 kdiff3、Eclipse 中的 diff/merge 工具、Understand 中的 diff/merge 工具等。

练习 1：实现 firstfit 连续物理内存分配算法（需要编程）。

在实现 firstfit 内存分配算法的回收函数时，要考虑地址连续的空闲块之间的合并操作。

提示：在建立空闲页块链表时，需要按照空闲页块起始地址来排序，形成一个有序的链表。可能会修改 default_pmm.c 中的 default_init、default_init_memmap、default_alloc_pages、default_free_pages 等相关函数。请仔细查看和理解 default_pmm.c 中的注释。

练习 2：实现寻找虚拟地址对应的页表项（需要编程）。

通过设置页表和对应的页表项，可建立虚拟内存地址和物理内存地址的对应关系。其中的 get_pte 函数是设置页表项环节中的一个重要步骤。此函数找到一个虚地址对应的二

级页表项的内核虚地址，如果此二级页表项不存在，则分配一个包含此项的二级页表。本练习需要补全位于 kern/mm/pmm.c 中的 get_pte 函数，实现其功能。请仔细查看和理解 get_pte 函数中的注释。get_pte 函数的调用关系图如图 3-1 所示。

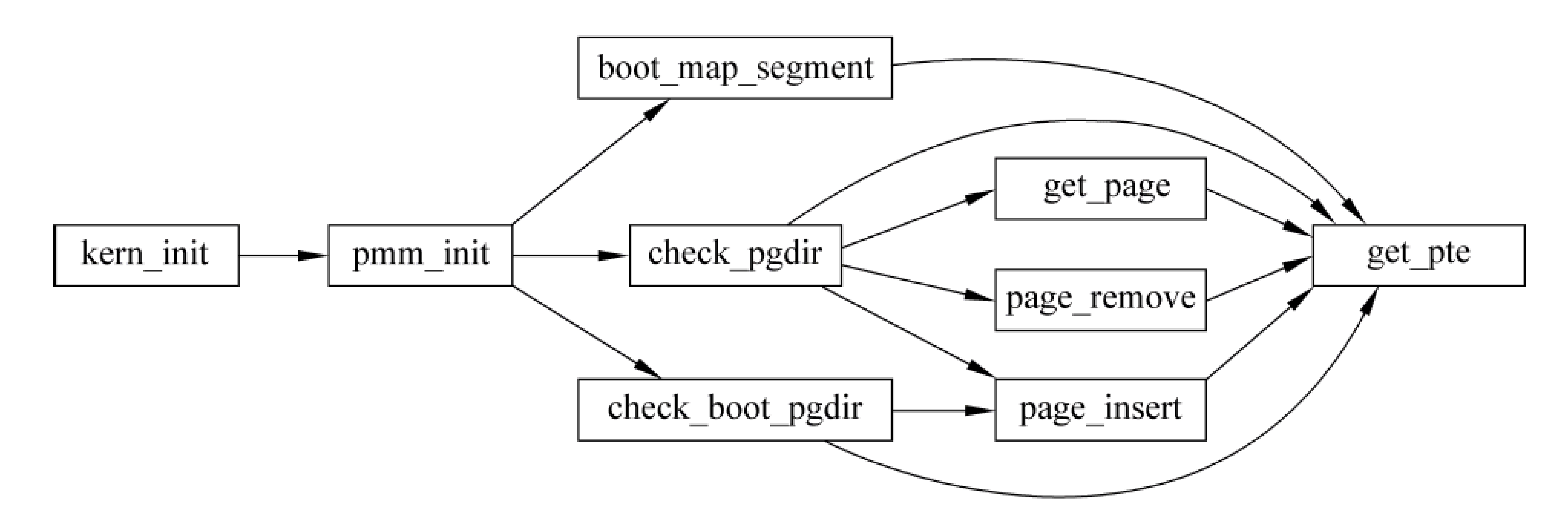


图 3-1　get_pte 函数的调用关系图

练习 3：释放某虚地址所在的页并取消对应二级页表项的映射(需要编程)。

当释放一个包含某虚地址的物理内存页时，需要让对应此物理内存页的管理数据结构 Page 做相关的清除处理，使得此物理内存页成为空闲；另外还需把表示虚地址与物理地址对应关系的二级页表项清除。请仔细查看和理解 page_remove_pte 函数中的注释。为此，需要补全在 kern/mm/pmm.c 中的 page_remove_pte 函数。page_remove_pte 函数的调用关系图如图 3-2 所示。

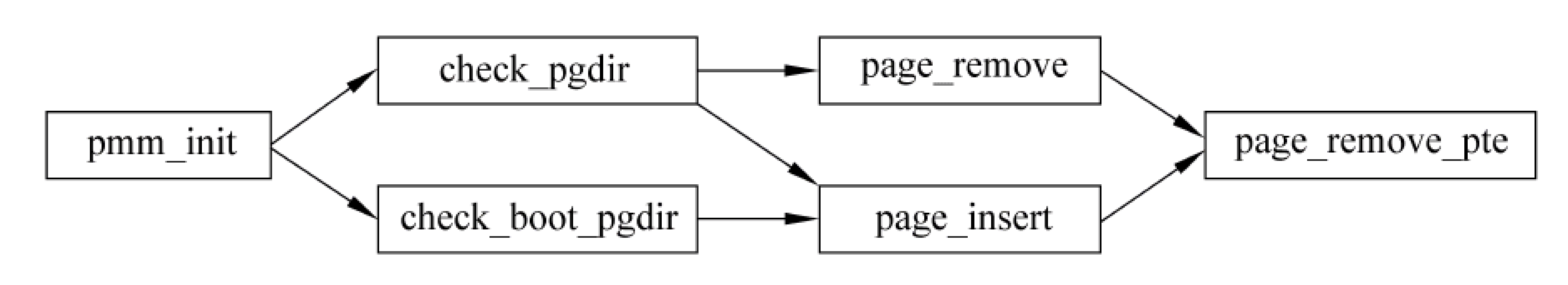


图 3-2　page_remove_pte 函数的调用关系图

扩展练习 Challenge：任意大小的内存单元 slub 分配算法(需要编程)。

如果觉得上述练习难度不够，可考虑完成此扩展练习。实现两层架构的高效内存单元分配，第一层是基于页大小的内存分配，第二层是在第一层基础上实现基于任意大小的内存分配。例如，如果连续分配 8 个 16B 的内存块，当分配完毕后，实际只消耗了一个空闲物理页。要求时空都高效，可参考 slub 算法来实现，可简化实现，能够体现其主体思想即可。要求有设计文档。slub 的相关网页是 http://www.ibm.com/developerworks/cn/linux/l-cn-slub/ 。完成 Challenge 的同学可单独提交 Challenge。

3.2.2　项目组成

目录结构图如图 3-3 所示。

相对与实验 1，实验 2 主要增加和修改的文件有 bootasm.S、entry.S、init.c、default_pmm.c、default_pmm.h、pmm.c、pmm.h、sync.h、atomic.h、list.h 和 kernel.ld。主要改动如下。

(1) boot/bootasm.S：增加了对计算机系统中物理内存布局的探测功能。

(2) kern/init/entry.S：根据临时段表重新暂时建立好新的段空间，为进行分页做好准备。

```
|-- boot
|   |-- asm.h
|   |-- bootasm.S
|   `-- bootmain.c
|-- kern
|   |-- init
|   |   |-- entry.S
|   |   `-- init.c
|   |-- mm
|   |   |-- default_pmm.c
|   |   |-- default_pmm.h
|   |   |-- memlayout.h
|   |   |-- mmu.h
|   |   |-- pmm.c
|   |   `-- pmm.h
|   |-- sync
|   |   `-- sync.h
|   `-- trap
|       |-- trap.c
|       |-- trapentry.S
|       |-- trap.h
|       `-- vectors.S
|-- libs
|   |-- atomic.h
|   |-- list.h
`-- tools
    |-- kernel.ld
```

图 3-3　目录结构图

(3) kern/mm/default_pmm.[ch]：提供基本的基于链表方法的物理内存管理(分配单位为页，即 4096B)。

(4) kern/mm/pmm.[ch]：pmm.h 定义物理内存管理类框架 struct pmm_manager，基于此通用框架可以实现不同的物理内存管理策略和算法(default_pmm.[ch] 实现了一个基于此框架的简单物理内存管理策略)；pmm.c 包含了对此物理内存管理类框架的访问，以及与建立、修改、访问页表相关的各种函数实现。

(5) kern/sync/sync.h：为确保内存管理修改相关数据时不被中断打断，提供两个功能，一个是保存 eflag 寄存器中的中断屏蔽位信息并屏蔽中断的功能，另一个是根据保存的中断屏蔽位信息来使能中断的功能。

(6) libs/list.h：定义了通用双向链表结构以及相关的查找、插入等基本操作，这是建立基于链表方法的物理内存管理(以及其他内核功能)的基础。其他有类似双向链表需求的内核功能模块可直接使用 list.h 中定义的函数。

(7) libs/atomic.h：定义了对一个变量进行读写的原子操作，确保相关操作不被中断打断。(可不用细看)

(8) tools/kernel.ld：ld 形成执行文件的地址所用到的链接脚本。修改了 ucore 的起始入口和代码段的起始地址。相关细节可参看附录 C。

编译并运行代码的命令如下：

```
make
make qemu
```

则可以得到如下显示(仅供参考)：

```
chenyu$ make qemu
(THU.CST) os is loading ...

Special kernel symbols:
    entry    0xc010002c (phys)
    etext    0xc010537f (phys)
    edata    0xc01169b8 (phys)
    end      0xc01178dc (phys)
Kernel executable memory footprint: 95KB
memory managment: default_pmm_manager
e820map:
    memory: 0009f400, [00000000, 0009f3ff], type=1.
    memory: 00000c00, [0009f400, 0009ffff], type=2.
    memory: 00010000, [000f0000, 000fffff], type=2.
    memory: 07efd000, [00100000, 07ffcfff], type=1.
    memory: 00003000, [07ffd000, 07ffffff], type=2.
    memory: 00040000, [fffc0000, ffffffff], type=2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
--------------------BEGIN--------------------
PDE(0e0) c0000000-f8000000 38000000 urw
    |-- PTE(38000) c0000000-f8000000 38000000-rw
PDE(001) fac00000-fb000000 00400000-rw
    |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
    |-- PTE(00001) fafeb000-fafec000 00001000-rw
---------------------END---------------------
++setup timer interrupts
100 ticks
100 ticks
⋮
```

通过上面显示，我们可以看到 ucore 在显示其 entry(入口地址)、etext(代码段截止处地址)、edata(数据段截止处地址)和 end(ucore 截止处地址)的值后，探测出计算机系统中的物理内存的布局(e820map 下的显示内容)。接下来 ucore 会以页为最小分配单位实现一个简单的内存分配管理，完成二级页表的建立，进入分页模式，执行各种我们设置的检查，最后显示 ucore 建好的二级页表内容，并在分页模式下响应时钟中断。

3.3 物理内存管理概述

3.3.1 实验执行流程概述

本次实验主要完成 ucore 内核对物理内存的管理工作。参考 ucore 总控函数 kern_init 的代码，可以清楚地看到在调用完成物理内存初始化的 pmm_init 函数之前和之后，是已有

lab1 实验的工作，好像没什么修改。其实不然，ucore 有两个方面的扩展。首先，bootloader 的工作有增加，在 bootloader 中，完成了对物理内存资源的探测工作（可进一步参阅本章附录 A 和附录 B），让 ucore kernel 在后续执行中能够基于 bootloader 探测出的物理内存情况进行物理内存管理初始化工作。其次，bootloader 不像 lab1 那样，直接调用 kern_init 函数，而是先调用位于 lab2/kern/init/entry.S 中的 kern_entry 函数。kern_entry 函数的主要任务是为执行 kern_init 建立一个良好的 C 语言运行环境（设置堆栈），而且临时建立了一个段映射关系，为之后建立分页机制的过程做一个准备（细节在 3.5.5 节有进一步阐述）。完成这些工作后，才调用 kern_init 函数。

kern_init 函数在完成一些输出并对 lab1 实验结果的检查后，将进入物理内存管理初始化的工作，即调用 pmm_init 函数完成物理内存的管理，这也是 lab2 的内容。接着是执行中断和异常相关的初始化工作，即调用 pic_init 函数和 idt_init 函数等，这些工作与 lab1 的中断异常初始化工作的内容是相同的。

为了完成物理内存管理，这里首先需要探测可用的物理内存资源；了解到物理内存位于什么地方、有多大之后，就以固定页面大小来划分整个物理内存空间，并准备以此为最小内存分配单位来管理整个物理内存，管理在内核运行过程中每页内存，设定其可用状态（free 的，used 的，还是 reserved 的），这其实就对应了连续内存分配概念和原理的具体实现；接着 ucore kernel 就要建立页表，启动分页机制，让 CPU 的 MMU 把预先建好的页表中的页表项读入到 TLB 中，根据页表项描述的虚拟页（Page）与物理页帧（Page Frame）的对应关系完成 CPU 对内存的读、写和执行操作。这一部分其实就对应了内存映射、页表、多级页表等概念和原理的具体实现。

在代码分析上，建议根据执行流程来直接看源代码，并可采用 GDB 源码调试的手段来动态地分析 ucore 的执行过程。内存管理相关的总体控制函数是 pmm_init 函数，它完成的主要工作包括如下。

(1) 初始化物理内存页管理器框架 pmm_manager。
(2) 建立空闲的 page 链表，这样就可以分配以页（4KB）为单位的空闲内存了。
(3) 检查物理内存页分配算法。
(4) 为确保切换到分页机制后，代码能够正常执行，先建立一个临时二级页表。
(5) 建立一一映射关系的二级页表。
(6) 使能分页机制。
(7) 重新设置全局段描述符表。
(8) 取消临时二级页表。
(9) 检查页表建立是否正确。
(10) 通过自映射机制完成页表的打印输出（这部分是扩展知识）。

另外，主要注意的相关代码内容包括如下。

① boot/bootasm.S 中探测内存部分（从 probe_memory 到 finish_probe 的代码）。

② 管理每个物理页的 Page 数据结构（在 mm/memlayout.h 中），这个数据结构也是实现连续物理内存分配算法的关键数据结构，可通过此数据结构来完成空闲块的链接和信息存储，而基于这个数据结构的管理物理页数组起始地址就是全局变量 pages，具体初始化此数组的函数位于 page_init 函数中。

③ 用于实现连续物理内存分配算法的物理内存页管理器框架 pmm_manager，这个数据结构定义了实现内存分配算法的关键函数指针，而同学需要完成这些函数的具体实现。

④ 设定二级页表和建立页表项以完成虚实地址映射关系，这与硬件相关，且用到不少内联函数，源代码相对难懂一些。具体完成页表和页表项建立的重要函数是 boot_map_segment 函数，而 get_pte 函数是完成虚实映射关键的关键。

3.3.2 探测系统物理内存布局

当 ucore 启动之后，最重要的事情就是知道还有多少内存可用。一般来说，获取内存大小的方法有 BIOS 中断调用和直接探测两种。BIOS 中断调用方法一般只能在实模式下完成，直接探测方法必须在保护模式下完成。通过 BIOS 中断获取内存布局有三种方式，都是基于 INT 15h 中断，分别为 88h、e801h、e820h。但是，并非在所有情况下这三种方式都能工作。在 Linux Kernel 里，采用的方法是依次尝试这三种方法。在本实验中，我们通过 e820h 中断获取内存信息。因为 e820h 中断必须在实模式下使用，所以在 bootloader 进入保护模式之前调用这个 BIOS 中断，并且把 e820 映射结构保存在物理地址 0x8000 处。具体实现详见 boot/bootasm.S。有关探测系统物理内存方法和具体实现的信息参见本章附录 A 和附录 B。

3.3.3 以页为单位管理物理内存

在获得可用物理内存范围后，系统需要建立相应的数据结构来管理以物理页（按 4KB 对齐，且大小为 4KB 的物理内存单元）为最小单位的整个物理内存，以配合后续涉及的分页管理机制。每个物理页可以用一个 Page 数据结构来表示。由于一个物理页需要占用一个 Page 结构的空间，Page 结构在设计时须尽可能小，以减少对内存的占用。Page 的定义在 kern/mm/memlayout.h 中。以页为单位的物理内存分配管理的实现在 kern/default_pmm.[ch]。

为了与以后的分页机制配合，首先需要建立对整个计算机的每一个物理页的属性，用结构 Page 来表示，它包含了映射此物理页的虚拟页个数，描述物理页属性的 flags 和双向链接各个 Page 结构的 page_link 双向链表。

```
struct Page {
    int ref;
    uint32_t flags;
    unsigned int property;
    list_entry_t page_link;
};
```

这里看看 Page 数据结构的各个成员变量有何具体含义。ref 表示该页被页表的引用记数（在“实现分页机制”一节会讲到）。如果这个页被页表引用了，即在某页表中有一个页表项设置了一个虚拟页到这个 Page 管理的物理页的映射关系，就会把 Page 的 ref 加 1；反之，若页表项取消，即映射关系解除，就会把 Page 的 ref 减 1。flags 表示此物理页的状态标记，进一步查看 kern/mm/memlayout.h 中的定义，可以看到：

```
/* Flags describing the status of a page frame */
```

```
#define PG_reserved  0  //the page descriptor is reserved for kernel or unusable
#define PG_property  1  //the member 'property' is valid
```

这表示 flags 目前用到了两个 bit 表示页目前具有的两种属性，bit 0 表示此页是否被保留，如果是被保留的页，则 bit 0 会设置为 1，且不能放到空闲页链表中，即这样的页不是空闲页，不能动态分配与释放。比如目前内核代码占用的空间就属于这样“被保留”的页。在本实验中，bit 1 表示此页是否是空闲的，如果设置为 1，表示这页是空闲的，可以被分配；如果设置为 0，表示这页已经被分配出去了，不能被再二次分配。另外，本实验这里取的名字 PG_property 不直观，主要是我们可以设计不同的页分配算法，那么这个 PG_property 就有不同的含义了。

在本实验中，Page 数据结构的成员变量 property 用来记录某连续内存空闲块的大小（即地址连续的空闲页的个数）。这里需要注意的是用到此成员变量的这个 Page 比较特殊，是这个连续内存空闲块地址最小的一页（即头一页，Head Page）。连续内存空闲块利用这个页的成员变量 property 来记录在此块内的空闲页的个数。这里取的名字 property 也不是很直观，原因与上面类似，在不同的页分配算法中，property 有不同的含义。

Page 数据结构的成员变量 page_link 是一个便于把多个连续内存空闲块链接在一起的双向链表指针（可回顾在 lab0 中有关双向链表数据结构的介绍）。这里需要注意的是用到此成员变量的这个 Page 比较特殊，是这个连续内存空闲块地址最小的一页（即头一页，Head Page）。连续内存空闲块利用这个页的成员变量 page_link 来链接比它地址小和大的其他连续内存空闲块。

在初始情况下，也许这个物理内存的空闲物理页都是连续的，这样就形成了一个大的连续内存空闲块。但随着物理页的分配与释放，这个大的连续内存空闲块会分裂为一系列地址不连续的多个小连续内存空闲块，且每个连续内存空闲块内部的物理页是连续的。为了有效地管理这些小连续内存空闲块，所有的连续内存空闲块可用一个双向链表来管理，便于分配和释放，为此定义了一个 free_area_t 数据结构，它包含了一个 list_entry 结构的双向链表指针和记录当前空闲页的个数的无符号整型变量 nr_free。其中的链表指针指向了空闲的物理页。

```
/* free_area_t-maintains a doubly linked list to record free (unused) pages */
typedef struct {
    list_entry_t free_list;           //the list header
    unsigned int nr_free;             //number of free pages in this free list
} free_area_t;
```

有了这两个数据结构，ucore 就可以管理起来整个以页为单位的物理内存空间。接下来需要解决两个问题。

（1）管理页级物理内存空间所需的 Page 结构的内存空间从哪里开始，占多大空间？

（2）空闲内存空间的起始地址在哪里？

对于这两个问题，我们首先根据 bootloader 给出的内存布局信息找出最大的物理内存地址 maxpa（定义在 page_init 函数中的局部变量），由于 x86 的起始物理内存地址为 0，所以可以得知需要管理的物理页个数为

```
npage=maxpa / PGSIZE
```

这样，就可以预估出管理页级物理内存空间所需的Page结构的内存空间所需的内存大小为

```
sizeof(struct Page) * npage)
```

由于bootloader加载ucore的结束地址（用全局指针变量end记录）以上的空间没有被使用，所以我们可以把end按页大小为边界取整后，作为管理页级物理内存空间所需的Page结构的内存空间，记为：

```
pages=(struct Page *)ROUNDUP((void *)end, PGSIZE);
```

为了简化起见，从地址0到地址pages+sizeof(struct Page) * npage)结束的物理内存空间设定为已占用物理内存空间（起始0～640KB的空间是空闲的），地址pages+sizeof(struct Page) * npage)以上的空间为空闲物理内存空间，这时的空闲空间起始地址为

```
uintptr_t freemem=PADDR((uintptr_t)pages+sizeof(struct Page) * npage);
```

为此我们需要把这两部分空间给标识出来。首先，对于所有物理空间，通过如下语句即可实现占用标记：

```
for (i=0; i<npage; i++) {
    SetPageReserved(pages+i);
}
```

然后，根据探测到的空闲物理空间，通过如下语句即可实现空闲标记：

```
//获得空闲空间的起始地址begin和结束地址end
    ⋮
    init_memmap(pa2page(begin), (end-begin)/PGSIZE);
```

其实SetPageReserved只需把物理地址对应的Page结构中的flags标志设置为PG_reserved，表示这些页已经被使用了，将来不能被用于分配。而init_memmap函数则是把空闲物理页对应的Page结构中的flags和引用计数ref清零，并加到free_area.free_list指向的双向列表中，为将来的空闲页管理做好初始化准备工作。

操作系统关于内存分配原理方面的知识有很多，但在本实验中只实现了最简单的内存页分配算法。相应的实现在default_pmm.c中的default_alloc_pages函数和default_free_pages函数，相关实现很简单，这里就不具体分析了，直接看源码，应该很好理解。

其实实验2在内存分配和释放方面最主要的作用是建立了一个物理内存页管理器框架，这实际上是一个函数指针列表，定义如下：

```
struct pmm_manager {
    const char * name;                                 //物理内存页管理器的名字
    void (* init)(void);                               //初始化内存管理器
    void(* init_memmap)(struct Page * base,size_t n);  //初始化管理空闲内存页的数据结构
    struct Page * (* alloc_pages)(size_t n);           //分配n个物理内存页
```

```
    void(* free_pages)(struct Page* base,size_t n);          //释放 n个物理内存页
    size_t (* nr_free_pages)(void);                          //返回当前剩余的空闲页数
    void (* check)(void);                         //用于检测分配/释放实现是否正确的辅助函数
};
```

重点是实现 init_memmap、alloc_pages、free_pages 这三个函数。当完成物理内存页管理初始化工作后，计算机系统的内存布局如图 3-4 所示。

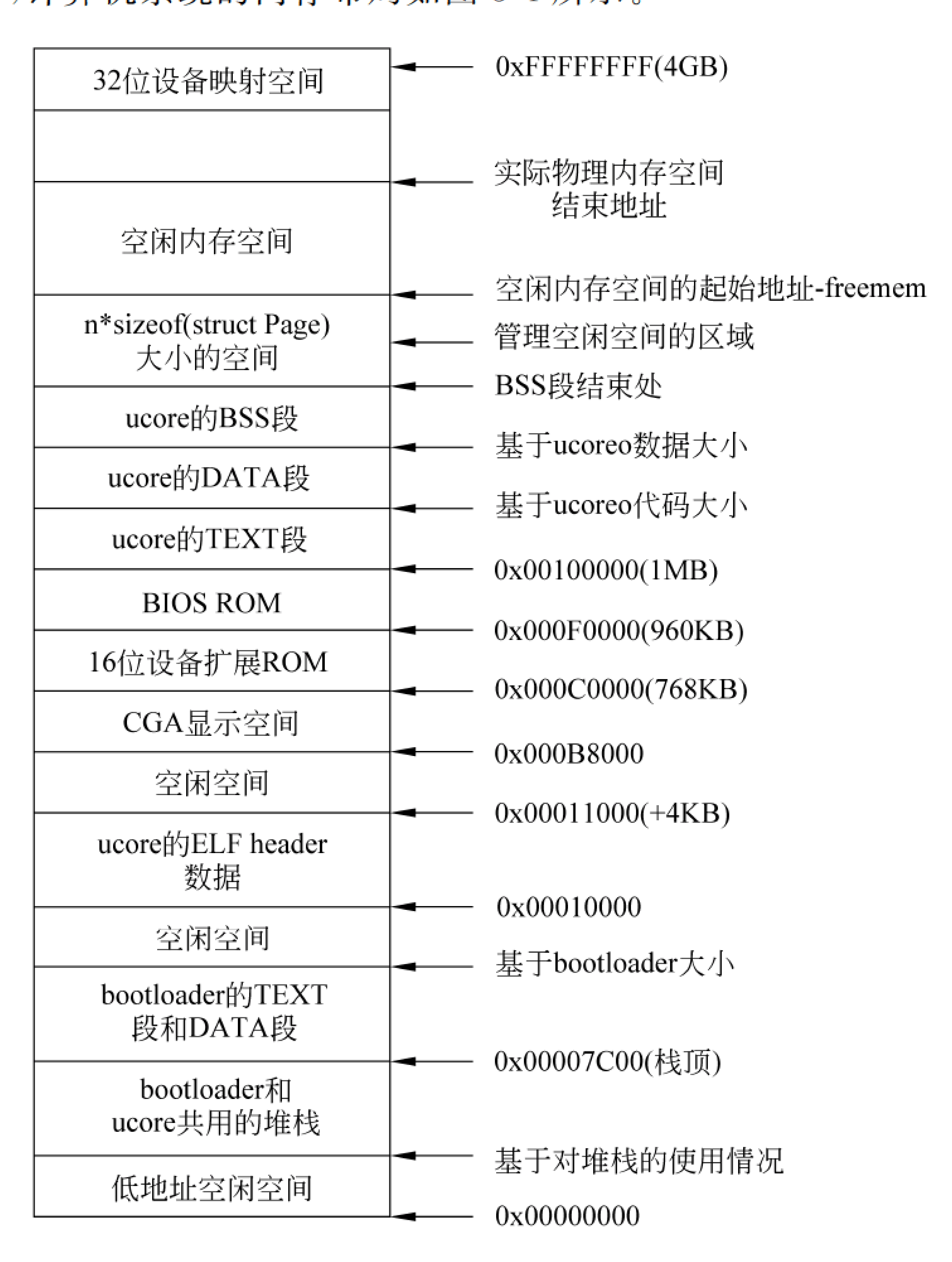


图 3-4　计算机系统的内存布局

3.3.4　物理内存页分配算法实现

如果要在 ucore 中实现连续物理内存分配算法，则需要考虑的问题比较多，相对课本上的物理内存分配算法描述要复杂不少。下面介绍一下实现一个 first_fit 内存分配算法的大致流程。

lab2 的第一部分是完成 first_fit 的分配算法。first_fit 内存分配算法原理上很简单，但要在 ucore 中实现，需要充分了解并利用 ucore 已有的数据结构和相关操作、关键的一些全局变量等。

1. 关键数据结构和变量

first_fit 分配算法需要维护一个查找有序(地址按从小到大排列)空闲块(以页为最小

单位的连续地址空间)的数据结构,而双向链表是一个很好的选择。

libs/list.h 定义了可挂接任意元素的通用双向链表结构和对应的操作,所以需要了解如何使用这个文件提供的各种函数,从而可以完成对双向链表的初始化、插入、删除等操作。

kern/mm/memlayout.h 中定义了一个 free_area_t 数据结构,包含成员结构如下:

```
list_entry_t free_list;          //空闲块双向链表的头
unsigned int nr_free;            //空闲块的总数(以页为单位)
```

显然,我们可以通过此数据结构来完成对空闲块的管理,而 default_pmm.c 中定义的 free_area 变量就是做这个事情的。

kern/mm/pmm.h 中定义了一个通用的分配算法的函数列表,用 pmm_manager 表示。其中 init 函数就是用来初始化 free_area 变量的, first_fit 分配算法可直接重用 default_init 函数的实现。init_memmap 函数需要根据现有的内存情况构建空闲块列表的初始状态。何时应该执行这个函数呢?

通过分析代码,可以知道:

```
kern_init-->pmm_init-->page_init-->init_memmap-->pmm_manager->init_memmap
```

所以,default_init_memmap 需要根据 page_init 函数中传递过来的参数(某个连续地址的空闲块的起始页,页个数)来建立一个连续内存空闲块的双向链表。这里有一个假定 page_init 函数是按地址从小到大的顺序传来的连续内存空闲块的。链表头是 free_area.free_list,链表项是 Page 数据结构的 base－>page_link。这样我们就依靠 Page 数据结构中的成员变量 page_link 形成了连续内存空闲块列表。

2. 设计实现

default_init_memmap 函数将根据每个物理页帧的情况来建立空闲页链表,且空闲页块应该根据地址高低形成一个有序链表。根据上述变量的定义,default_init_memmap 可大致实现如下:

```
default_init_memmap(struct Page * base, size_t n) {
    struct Page * p=base;
    for (; p !=base+n; p++) {
        p->flags=p->property=0;
        set_page_ref(p, 0);
    }
    base->property=n;
    SetPageProperty(base);
    nr_free+=n;
    list_add(&free_list, &(base->page_link));
}
```

如果要分配一个页,需要考虑哪些问题呢?这里就需要考虑实现 default_alloc_pages 函数,注意参数 n 表示要分配 n 个页。另外,需要注意实现时尽量多考虑一些边界情况,这样确保软件的鲁棒性。例如:

```
if (n>nr_free) {
```

```
        return NULL;
    }
```

这样可以确保分配不会超出范围。也可加一些 assert 函数，当有错误出现时，能够迅速发现。例如，n 应该大于 0，我们就可以加上

```
    assert(n>0);
```

这样在 n<=0 的情况下，ucore 会迅速报错。first_fit 需要从空闲链表头开始查找最小的地址，通过 list_next 找到下一个空闲块元素，通过 le2page 宏可以更加链表元素获得对应的 Page 指针 p。通过 p->property 可以了解此空闲块的大小。如果>=n，这就找到了！如果<n，则通过 list_next，继续查找。直到 list_next== &free_list，这表示找完一遍了。找到后，就要重新组织空闲块，然后把找到的 page 返回。所以 default_alloc_pages 可大致实现如下：

```
    static struct Page *
    default_alloc_pages(size_t n) {
        if (n>nr_free) {
            return NULL;
        }
        struct Page * page=NULL;
        list_entry_t * le=&free_list;
        while ((le=list_next(le)) !=&free_list) {
            struct Page * p=le2page(le, page_link);
            if (p->property>=n) {
                page=p;
                break;
            }
        }
        if (page !=NULL) {
            list_del(&(page->page_link));
            if (page->property>n) {
                struct Page * p=page+n;
                p->property=page->property-n;
                list_add(&free_list, &(p->page_link));
            }
            nr_free-=n;
            ClearPageProperty(page);
        }
        return page;
    }
```

default_free_pages 函数的实现其实是 default_alloc_pages 的逆过程，不过需要考虑空闲块的合并问题。这里就不再细讲了。注意，上述代码只是参考设计，不是完整的正确设计。更详细的说明位于 lab2/kernel/mm/default_pmm.c 的注释中。希望读者能够顺利完成本实验的第一部分。

3.3.5 实现分页机制

1. 段页式管理基本概念

在保护模式中，x86 体系结构将内存地址分成三种：逻辑地址（也称虚地址）、线性地址和物理地址。逻辑地址就是程序指令中使用的地址，物理地址是实际访问内存的地址。逻辑地址通过段式管理的地址映射可以得到线性地址，线性地址通过页式管理的地址映射得到物理地址。段页式管理总体框架图如图 3-5 所示。

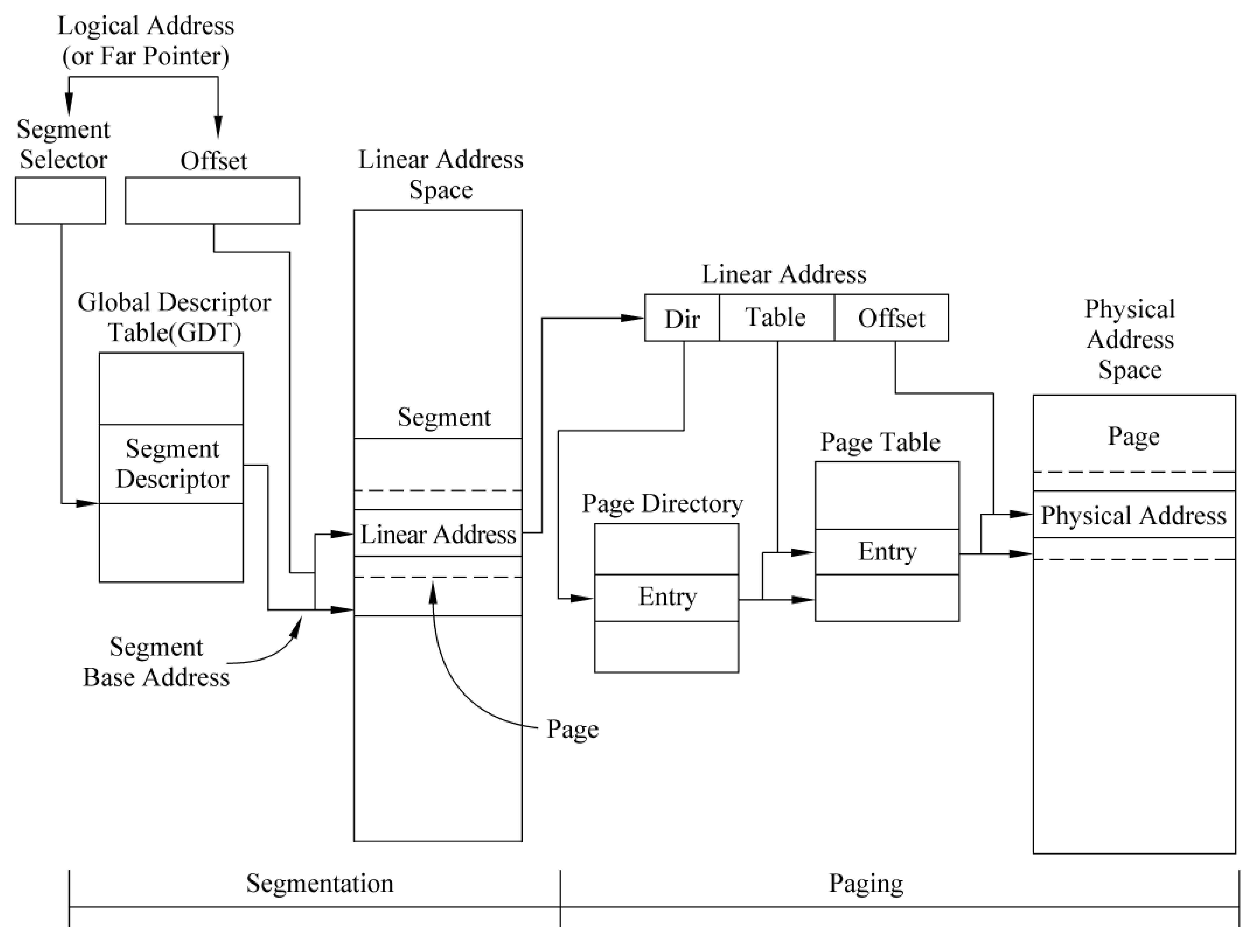


图 3-5 段页式管理总体框架图

段式管理前一个实验已经讨论过。在 ucore 中段式管理只起到了一个过渡作用，它将逻辑地址不加转换直接映射成线性地址，所以我们在下面的讨论中可以对这两个地址不加区分（目前的 OS 实现也是不加区分的）。对段式管理有兴趣的同学可以参考《Intel 64 and IA-32Architectures Software Developer's Manual-Volume 3A》3.2 节。

如图 3-6 所示，页式管理将线性地址分成三部分，即 Directory 部分、Table 部分和 Offset 部分）。ucore 的页式管理通过一个二级的页表实现。一级页表的起始物理地址存放在 CR3 寄存器中，这个地址必须是一个页对齐的地址，也就是低 12 位必须为 0。目前，ucore 用 boot_cr3（位于 mm/pmm.c 中）记录这个值。

2. 建立段页式管理中需要考虑的关键问题

为了实现分页机制，需要建立好虚拟内存和物理内存的页映射关系，即正确建立二级页表。此过程涉及硬件细节，不同的地址映射关系组合，相对比较复杂。总体而言，我们需要思考如下问题。

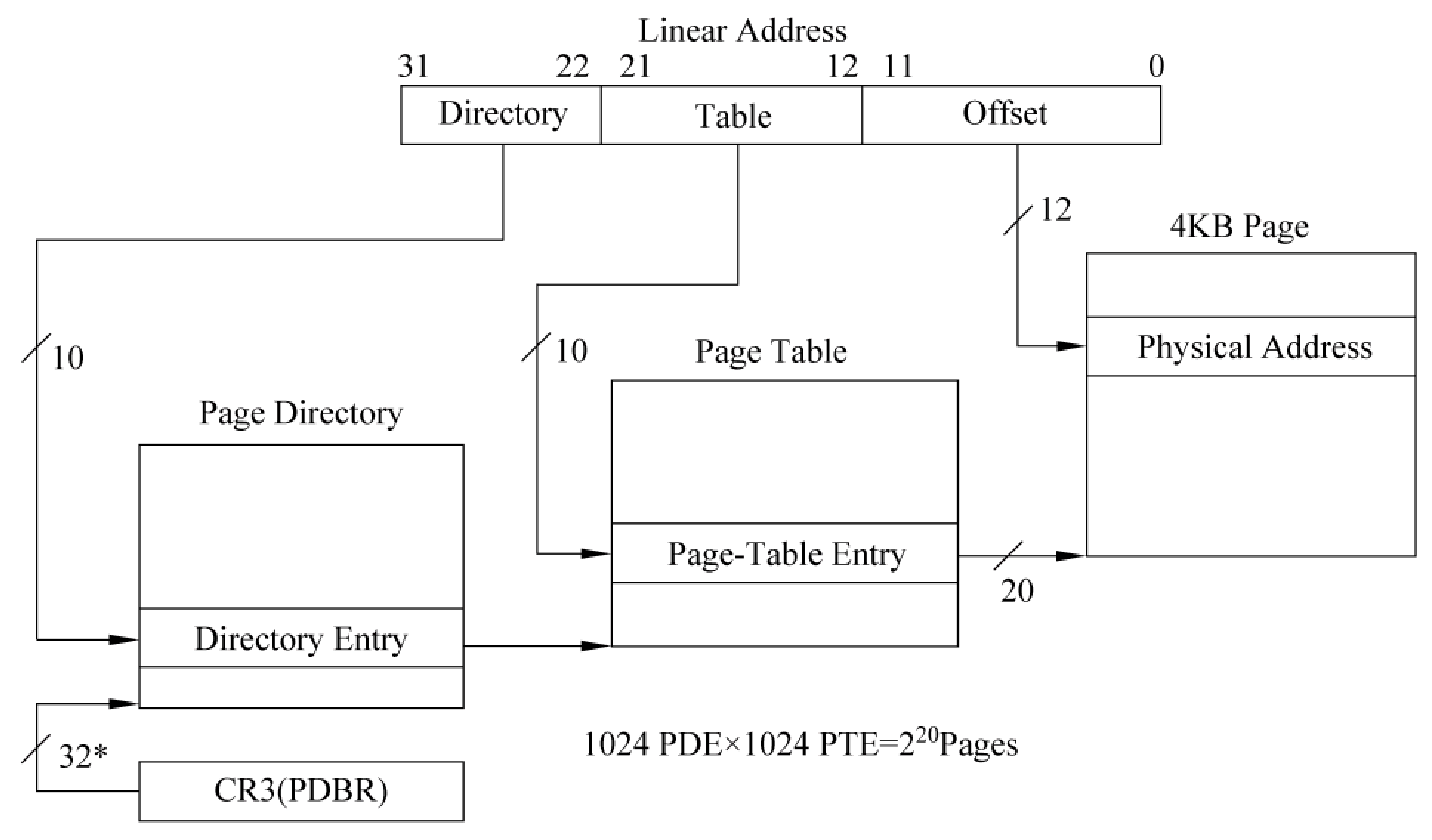


图 3-6　分页机制管理

（1）如何在建立页表的过程中维护全局段描述符表（GDT）和页表的关系，确保 ucore 能够在各个时间段上都能正常寻址？

（2）对于哪些物理内存空间需要建立页映射关系？

（3）具体的页映射关系是什么？

（4）页目录表的起始地址设置在哪里？

（5）页表的起始地址设置在哪里，需要多大空间？

（6）如何设置页目录表项的内容？

（7）如何设置页表项的内容？

3. 建立虚拟页和物理页帧的地址映射关系

1）从链接脚本分析 ucore 执行时的地址

首先观察一下 tools/kernel.ld 文件在 lab1 和 lab2 中的区别。在 lab1 中：

```
ENTRY(kern_init)

SECTIONS {
    /* Load the kernel at this address: "." means the current address */
    . = 0x100000;

    .text : {
        * (.text .stub .text.* .gnu.linkonce.t.* )
    }
```

这意味着在 lab1 中通过 ld 工具形成的 ucore 的起始虚拟地址从 0x100000 开始。

注意：这个地址是虚拟地址。但由于 lab1 中建立的段地址映射关系为对等关系，所以 ucore 的物理地址也是 0x100000。而入口函数为 kern_init 函数。在 lab2 中：

```
ENTRY(kern_entry)
```

```
SECTIONS {
    /* Load the kernel at this address: "." means the current address */
    .= 0xC0100000;

    .text : {
        * (.text .stub .text.* .gnu.linkonce.t.* )
    }
```

这意味着 lab2 中通过 ld 工具形成的 ucore 的起始虚拟地址从 0xC0100000 开始。

注意：这个地址也是虚拟地址。入口函数为 kern_entry 函数。这与 lab1 有很大差别。但其实在 lab1 和 lab2 中，bootloader 把 ucore 都放在了起始物理地址为 0x100000 的物理内存空间。这实际上说明了 ucore 在 lab1 和 lab2 中采用的地址映射不同。

lab1：Virtual Address＝Linear Address＝Physical Address。

lab2：Virtual Address＝Linear Address＝Physical Address＋0xC0000000。

lab1 只采用了段映射机制，但在 lab2 中，启动好分页管理机制后，形成的是段页式映射机制，从而使得虚拟地址空间和物理地址空间之间存在如下映射关系：

```
Virtual Address= Linear Address= 0xC0000000+ Physical Address
```

另外，ucore 的入口地址也改为 kern_entry 函数，这个函数位于 init/entry.S 中，分析代码可以看出，entry.S 重新建立了段映射关系，从以前的

```
Virtual Address= Linear Address
```

改为

```
Virtual Address= Linear Address- 0xC0000000
```

由于 gcc 编译出的虚拟起始地址从 0xC0100000 开始，ucore 被 bootloader 放置在从物理地址 0x100000 处开始的物理内存中。所以当 kern_entry 函数完成新的段映射关系后，且 ucore 在没有建好页映射机制前，CPU 按照 ucore 中的虚拟地址执行，能够被分段机制映射到正确的物理地址上，确保 ucore 运行正确。

由于物理内存页管理器管理了从 0 到实际可用物理内存大小的物理内存空间，所以对于这些物理内存空间都需要建好页映射关系。由于目前 ucore 只运行在内核空间，所以可以建立一个一一映射关系。假定内核虚拟地址空间的起始地址为 0xC0000000，则虚拟内存和物理内存的具体页映射关系为

```
Virtual Address= Physical Address+ 0xC0000000
```

2）建立二级页表

由于已经具有了一个物理内存页管理器 default_pmm_manager，我们就可以用它来获得所需的空闲物理页。在二级页表结构中，页目录表占 4KB 空间，ucore 就可通过 default_pmm_manager 的 default_alloc_pages 函数获得一个空闲物理页，这个页的起始物理地址就是页目录表的起始地址。同理，ucore 也通过这种方式获得各个页表所需的空间。页表的空间大小取决与页表要管理的物理页数 n，一个页表项(32 位，即 4B)可管理一个物理页，页

表需要占 $n/256$ 个物理页空间。这样页目录表和页表所占的总大小为 $4096+1024\times n$B。

为把 0～KERNSIZE（明确 ucore 设定实际物理内存不能超过 KERNSIZE 值，即 0x38000000B，896MB，3670016 个物理页）的物理地址一一映射到页目录表项和页表项的内容，其大致流程如下。

(1) 先通过 default_pmm_manager 获得一个空闲物理页，用于页目录表。

(2) 调用 boot_map_segment 函数建立一一映射关系，具体处理过程以页为单位进行设置，即 Virtual Address＝Physical Address＋0xC0000000。

设一个逻辑地址 la（按页对齐，故低 12 位为零）对应的物理地址为 pa（按页对齐，故低 12 位为零），如果在页目录表项（la 的高 10 位为索引值）中的存在位（PTE_P）为 0，表示缺少对应的页表空间，则可通过 default_pmm_manager 获得一个空闲物理页给页表，页表起始物理地址是按 4096B 对齐的，这样填写页目录表项的内容为

```
页目录表项内容=页表起始物理地址|PTE_U|PTE_W|PTE_P
```

进一步对于页表中对应页表项（la 的中 10 位为索引值）的内容为

```
页表项内容=pa|PTE_P|PTE_W
```

其中各项含义如下。

① PTE_U：位 3，表示用户态的软件可以读取对应地址的物理内存页内容。

② PTE_W：位 2，表示物理内存页内容可写。

③ PTE_P：位 1，表示物理内存页存在。

ucore 的内存管理经常需要查找页表：给定一个虚拟地址，找出这个虚拟地址在二级页表中对应的项。通过更改此项的值可以方便地将虚拟地址映射到另外的页上。可完成此功能的这个函数是 get_pte 函数。它的原型为

```
pte_t* get_pte (pde_t* pgdir, uintptr_t la, bool create)
```

如图 3-7 所示的调用关系图可以比较好地看出 get_pte 在实现上述流程中的位置。

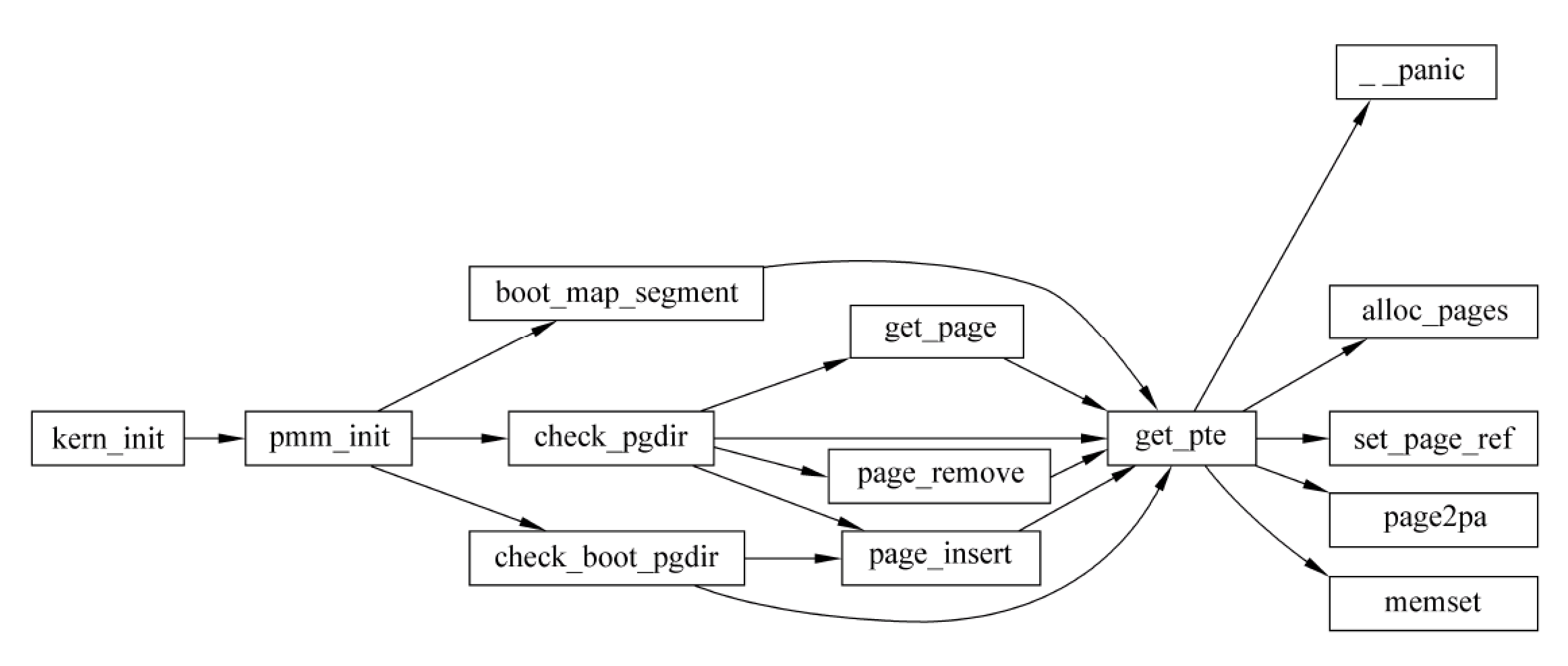


图 3-7 get_pte 调用关系图

这里涉及三个类型 pte_t、pde_t 和 uintptr_t。通过参见 mm/mmlayout.h 和 libs/types.h，可知它们其实都是 unsigned int 类型。在此做区分，是为了分清概念。

pde_t 全称为 page directory entry，也就是一级页表的表项（注意：pgdir 实际不是表项，而是一级页表本身。实际上应该新定义一个类型 pgd_t 来表示一级页表本身）。pte_t 全称为 page table entry，表示二级页表的表项。uintptr_t 表示为线性地址，由于段式管理只做直接映射，所以它也是逻辑地址。

pgdir 给出页表起始地址。通过查找这个页表，我们需要给出二级页表中对应项的地址。虽然目前只有 boot_pgdir 一个页表，但是引入进程的概念之后每个进程都会有自己的页表。

有可能根本就没有对应的二级页表的情况，所以二级页表不必一开始就分配，而是等到需要的时候再添加对应的二级页表。如果在查找二级页表项时，发现对应的二级页表不存在，则需要根据 create 参数的值来处理是否创建新的二级页表。如果 create 参数为 0，则 get_pte 返回 NULL；如果 create 参数不为 0，则 get_pte 需要申请一个新的物理页（通过 alloc_page 来实现，可在 mm/pmm.h 中找到它的定义），再在一级页表中添加页目录表项指向表示二级页表的新物理页。注意，新申请的页必须全部设定为零，因为这个页所代表的虚拟地址都没有被映射。

当建立从一级页表到二级页表的映射时，需要注意设置控制位。这里应该设置同时设置上 PTE_U、PTE_W 和 PTE_P（定义可在 mm/mmu.h）。如果原来就有二级页表，或者新建立了页表，则只需返回对应项的地址即可。

虚拟地址只有映射上了物理页才可以正常的读写。在完成映射物理页的过程中，除了要像上面那样在页表的对应表项上填上相应的物理地址外，还要设置正确的控制位。有关 x86 中页表控制位的详细信息，请参照《Intel 64 and IA-32 Architectures Software Developer's Manual-Volume 3A》4.11 节。

只有当一级二级页表的项都设置了用户写权限后，用户才能对对应的物理地址进行读写。所以可以在一级页表先给用户写权限，再在二级页表上面根据需要限制用户的权限，对物理页进行保护。由于一个物理页可能被映射到不同的虚拟地址上（如一块内存在不同进程间共享），当这个页需要在一个地址上解除映射时，操作系统不能直接把这个页回收，而是要先看看它还有没有映射到别的虚拟地址上。这是通过查找管理该物理页的 Page 数据结构的成员变量 ref（用来表示虚拟页到物理页的映射关系的个数）来实现的，如果 ref 为 0，表示没有虚拟页到物理页的映射关系，就可以把这个物理页给回收，从而这个物理页是空闲页，可以再被分配。page_insert 函数将物理页映射在页表上。可参看 page_insert 函数的实现来了解 ucore 内核是如何维护这个变量的。当不需要再访问这块虚拟地址时，可以把这块物理页回收并在将来用在其他地方。取消映射由 page_remove 来做，这其实是 page insert 的逆操作。

建立好一一映射的二级页表结构后，接下来就要使能分页机制了，这主要是通过 enable_paging 函数实现，这个函数主要做了两件事。

① 通过 lcr3 指令把页目录表的起始地址存入 CR3 寄存器中。

② 通过 lcr0 指令把 cr0 中的 CR0_PG 标志位设置上。

执行完 enable_paging 函数后，计算机系统便进入了分页模式。但到这一步还不够，还记得 ucore 在最开始通过 kern_entry 函数设置了临时的新段映射机制吗？这个临时的新段映射机制不是最简单的对等映射，导致虚拟地址和线性地址不相等。刚才建立的页映射关

系是建立在简单的段对等映射,即虚拟地址=线性地址的假设基础之上的。所以需要进一步调整段映射关系,即重新设置新的 GDT,建立对等段映射。

这里需要注意:在进入分页模式到重新设置新 GDT 的过程是一个过渡过程。在这个过渡过程中,已经建立了页表机制,所以通过现在的段机制和页机制实现的地址映射关系为

```
Virtual Address=Linear Address+0xC0000000=Physical Address+0xC0000000+0xC0000000
```

在这个特殊的阶段,如果不把段映射关系改为 Virtual Address=Linear Address,则通过段页式两次地址转换后,无法得到正确的物理地址。为此需要进一步调用 gdt_init 函数,根据新的 gdt 全局段描述符表内容(gdt 定义位于 pmm.c 中),恢复以前的段映射关系,即使得 Virtual Address=Linear Address。这样在执行完 gdt_init 后,通过的段机制和页机制实现的地址映射关系为:

```
Virtual Address=Linear Address=Physical Address+0xC0000000
```

这里存在的一个问题是,在调用 enable_page 函数使能分页机制后到执行完毕 gdt_init 函数重新建立好段页式映射机制的过程中,内核使用的还是旧的段表映射,也就是说,enable paging 之后,内核使用的是页表的低地址 entry。如何保证此时内核依然能够正常工作呢?其实只需让低地址目录表项的内容等于以 KERNBASE 开始的高地址目录表项的内容即可。目前内核大小不超过 4MB(实际上是 3MB,因为内核从 0x100000 开始编址),这样就只需要让页表在 0~4MB 的线性地址与 KERNBASE ~ KERNBASE+4MB 的线性地址获得相同的映射即可,都映射到 0~4MB 的物理地址空间,具体实现在 pmm.c 中 pmm_init 函数的语句:

```
boot_pgdir[0]=boot_pgdir[PDX(KERNBASE)];
```

实际上这种映射也限制了内核的大小。当内核大小超过预期的 3MB 就可能导致打开分页之后内核 crash,在后面的实验中,也的确出现了这种情况。解决方法同样简单,就是复制更多的高地址项到低地址。

当执行完 gdt_init 函数后,新的段页式映射已经建好了,上面的 0~4MB 的线性地址与 0~4MB 的物理地址一一映射关系已经没有用了。所以可以通过如下语句解除这个旧的映射关系。

```
boot_pgdir[0]=0;
```

在 page_init 函数建立完实现物理内存一一映射及页目录表自映射的页目录表和页表后,一旦使能分页机制,则 ucore 看到的内核虚拟地址空间如图 3-8 所示。

4. 不同运行阶段的地址映射关系

在大多数课本中,描述了基于段的映射关系、基于页的映射关系以及基于段页式的映射关系和 CPU 访存时对应的地址转换过程。但很少涉及操作系统是如何一步一步建立这个映射关系的。其实,在 lab1 和 lab2 中都会涉及如何建立映射关系的操作。在 lab1 中,我们已经碰到到了简单的段映射,即对等映射关系,保证了物理地址和虚拟地址相等,也就是通过建立全局段描述符表,让每个段的基址为 0,从而确定了对等映射关系。

在 lab2 中,由于在段地址映射的基础上进一步引入了页地址映射,形成了组合式的段

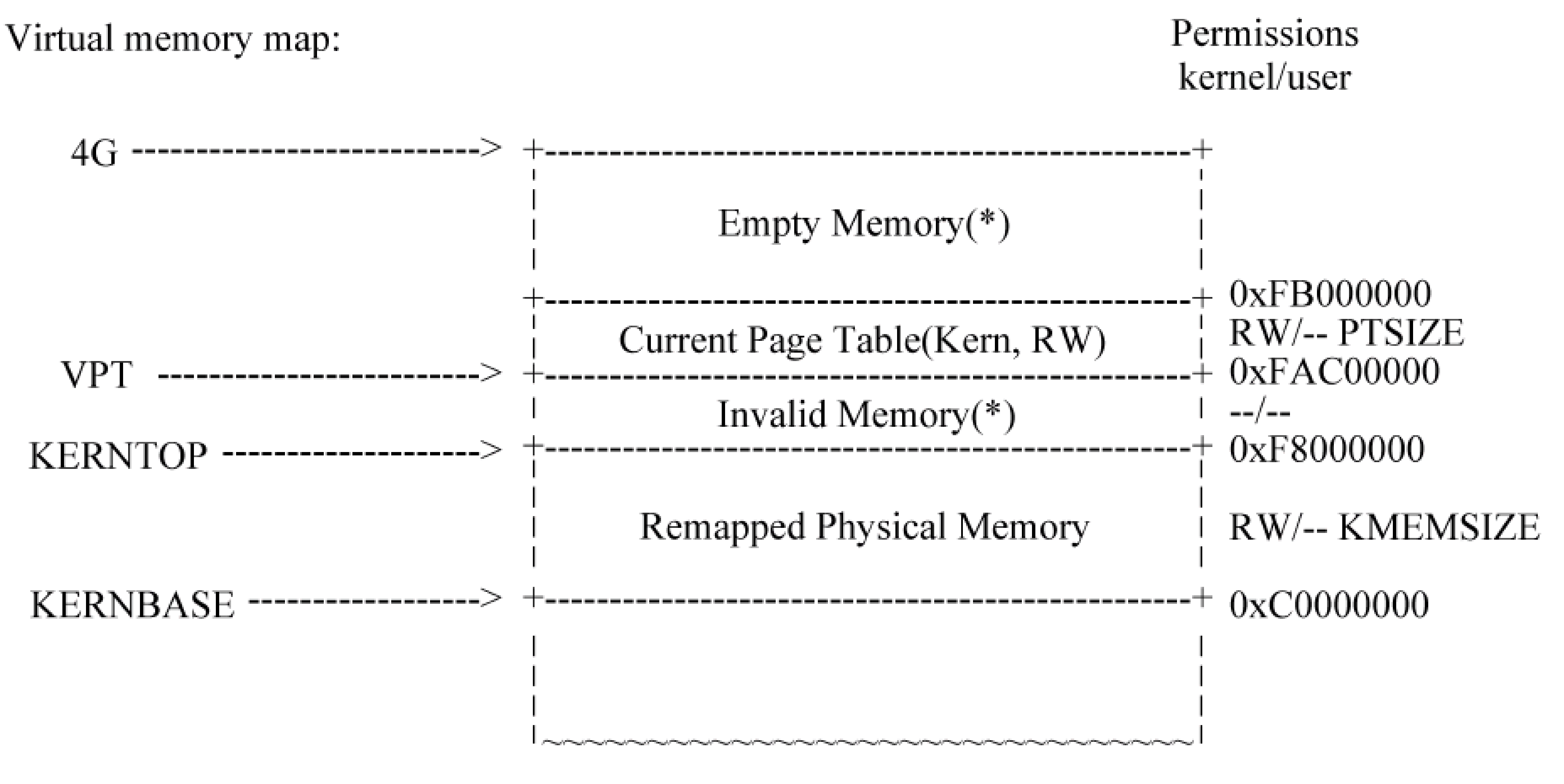


图 3-8　使能分页机制后的虚拟地址空间图

页式地址映射。这种方式虽然更加灵活,但实现的复杂性也增加了。在 lab2 中,ucore 从计算机加电,启动段式管理机制,启动段页式管理机制,在段页式管理机制下运行这整个过程中,虚地址到物理地址的映射产生了多次变化,接下来我们会逐一进行说明。

首先是 bootloader 地址映射阶段,bootloader 完成了与 lab1 一样的工作,即建立了基于段的对等映射(请查看 lab2/boot/bootasm.S 中的 finish_probe 地址处)。

接着进入了 ucore 启动页机制前的地址映射阶段,ucore 建立了一个一一段映射关系,其中虚拟地址 = 物理地址+0xC0000000(请查看 lab2/kern/init/entry.S 中的 kern_entry 函数)。

再接下来是建立并使能页表的临时段页式地址映射阶段,页表要表示的是线性地址与物理地址的对应关系为线性地址=物理地址+0xC0000000。这里有一个小技巧,让在 0～4MB的线性地址区域空间的线性地址(0～4MB)对应的物理地址=线性地址(0xC0000000～0xC0000000+4MB)对应的物理地址,这是通过 lab2/kern/mm/pmm.c 中第 321 行的代码实现的:

```
boot_pgdir[0]=boot_pgdir[PDX(KERNBASE)];
```

注意:此时 CPU 在寻址时只采用了分段机制。最后并使能分页映射机制(请查看 lab2/kern/mm/pmm.c 中的 enable_paging 函数),一旦执行完 enable_paging 函数中的加载 cr0 指令(即让 CPU 使能分页机制),则接下来的访问是基于段页式的映射关系了。对于(0xC0000000～0xC0000000+4MB)这块虚拟地址空间,最终会映射到哪些物理地址空间中呢?

由于段映射关系没有改变,使得经过段映射机制,虚拟地址范围(0xC0000000～0xC0000000+4MB)对应的线性地址=(0～4MB)。根据页表建立过程的描述,我们可知道线性地址空间(0～4MB)与线性地址空间(0xC0000000～0xC0000000+4MB)对应同样的物理地址,而线性地址空间(0xC0000000～0xC0000000+4MB)对应的物理地址空间为 0～4MB。这样对于(0xC0000000～0xC0000000+4MB)这块虚拟地址空间,段页式的地址映射关系为虚拟地址=线性地址+0xC0000000=物理地址+0xC0000000。

注意:这只是针对 0xC0000000～0xC0000000+4MB 这块虚拟地址空间。如果是

0xD0000000～0xD0000000＋4MB 这块虚拟地址空间，则段页式的地址映射关系为虚拟地址＝线性地址＋0xC0000000＝物理地址＋0xC0000000＋0xC0000000。这不是我们需要的映射关系，所以 0xC0000000＋4MB 以上的虚拟地址访问会出页错误异常。

最后一步完成收尾工作的正常段页式地址映射阶段，即首先调整段映射关系，这是通过加载新的全局段描述符表（pmm_init 函数调用 gdt_init 函数来完成）实现，这时的段映射关系为虚拟地址＝线性地址。然后通过执行语句“boot_pgdir[0]＝0;”把 boot_pgdir[0]的第一个页目录表项（0～4MB）清零来取消临时的页映射关系。至此，新的段页式的地址映射关系为虚拟地址 ＝ 线性地址 ＝物理地址＋0xC0000000。这也形成了 ucore 操作系统的内核虚拟地址空间的段页式映射关系，即虚拟地址空间（KERNBASE～KERNBASE＋KMEMSIZE）＝ 线性地址空间（KERNBASE～KERNBASE＋KMEMSIZE） ＝ 物理地址空间（0～KMEMSIZE）。

3.3.6 自映射机制

这是扩展知识。上一小节讲述了通过 boot_map_segment 函数建立了基于一一映射关系的页目录表项和页表项，这里的映射关系为

```
Virtual Address(KERNBASE~ KERNBASE+ KMEMSIZE)= Physical Address(0~ KMEMSIZE)
```

这样只要给出一个虚拟地址和一个物理地址，就可以设置相应 PDE 和 PTE，就可完成正确的映射关系。

如果我们这时需要按虚拟地址的地址顺序显示整个页目录表和页表的内容，则要查找页目录表的页目录表项内容，根据页目录表项内容找到页表的物理地址，再转换成对应的虚拟地址，然后访问页表的虚拟地址，搜索整个页表的每个页目录项。这样过程比较烦琐。

我们需要有一个简洁的方法来实现这个查找。ucore 做了一个很巧妙的地址自映射设计，把页目录表和页表放在一个连续的 4MB 虚拟地址空间中，并设置页目录表自身的虚地址与物理地址映射关系。这样在已知页目录表起始虚地址的情况下，通过连续扫描这特定的 4MB 虚拟地址空间，就很容易访问每个页目录表项和页表项内容。

具体而言，ucore 是这样设计的，首先设置了一个常量（见 memlayout.h）：

VPT＝0xFAC00000，这个地址的二进制表示为

```
1111 1010 1100 0000 0000 0000 0000 0000
```

高 10 位为 1111 1010 11，即十进制的 1003，中间 10 位为 0，低 12 位也为 0。在 pmm.c 中有两个全局初始化变量。

```
pte_t* const vpt= (pte_t* )VPT;
pde_t* const vpd= (pde_t* )PGADDR(PDX(VPT), PDX(VPT), 0);
```

并在 pmm_init 函数执行了如下语句：

```
boot_pgdir[PDX(VPT)]= PADDR(boot_pgdir)|PTE_P|PTE_W;
```

这些变量和语句有何特殊含义呢？其实 vpd 变量的值就是页目录表的起始虚拟地址 0xFAFEB000，且它的高 10 位和中 10 位是相等的，都是十进制的 1003。当执行了上述语

句，就确保了 vpd 变量的值就是页目录表的起始虚拟地址，且 vpt 是页目录表中第一个目录表项指向的页表的起始虚拟地址。此时描述内核虚拟空间的页目录表的虚地址为 0xFAFEB000，大小为 4KB。页表的理论连续虚拟地址空间 0xFAC00000～0xFB000000，大小为 4MB。因为这个连续地址空间的大小为 4MB，可有 1M 个 PTE，即可映射 4GB 的地址空间。

但 ucore 实际上不会用完这么多项，在 memlayout.h 中定义了常量：

```
#define KMEMSIZE        0x38000000
```

表示 ucore 只支持 896MB 的物理内存空间，这个 896MB 只是一个设定，可以根据情况改变，则最大的内核虚地址为常量：

```
#define KERNTOP        (KERNBASE+ KMEMSIZE)= 0xF8000000
```

所以最大内核虚地址 KERNTOP 的页目录项虚拟地址为

```
vpd+ 0xF8000000/0x400000= 0xFAFEB000+ 0x3E0= 0xFAFEB3E0
```

最大内核虚拟地址 KERNTOP 的页表项虚地址为

```
vpt+ 0xF8000000/0x1000= 0xFAC00000+ 0xF8000= 0xFACF8000
```

在 pmm.c 中的函数 print_pgdir 就是基于 ucore 的页表自映射方式完成了对整个页目录表和页表的内容扫描和打印。注意，这里不会出现某个页表的虚拟地址与页目录表虚拟地址相同的情况。

print pgdir 函数使得 ucore 具备和 qemu 的 info pg 相同的功能，即 print pgdir 能够从内存中将当前页表内有效数据(PTE_P)打印出来。复制出的格式如下：

```
PDE(0e0)    c0000000- f8000000    38000000    urw
|-- PTE(38000) c0000000- f8000000    38000000- rw
PDE(001)    fac00000- fb000000    00400000 - rw
|-- PTE(000e0)    faf00000- fafe0000    000e0000    urw
|-- PTE(00001)    fafeb000- fafec000    00001000 - rw
```

上面中的数字(包括括号里的)都是十六进制。

主要的功能是从页表中将具备相同权限的 PDE 和 PTE 项目组织起来。例如：

```
PDE(0e0) c0000000- f8000000 38000000 urw
```

(1) PDE(0e0)：0e0 表示 PDE 表中相邻的 224 项具有相同的权限。

(2) c0000000-f8000000：表示 PDE 表中，这相邻的两项所映射的线性地址的范围。

(3) 38000000：同样表示范围，即 f8000000 减去 c0000000 的结果。

(4) urw：PDE 表中所给出的权限位，u 表示用户可读，即 PTE_U，r 表示 PTE_P，w 表示用户可写，即 PTE_W。

```
PDE(001) fac00000- fb000000 00400000 - rw
```

表示仅 1 条连续的 PDE 表项具备相同的属性。相应地，在这条表项中遍历找到 2 组 PTE 表项，输出如下：

```
|--PTE(000e0)    faf00000-fafe0000   000e0000   urw
|--PTE(00001)    fafeb000-fafec000   00001000 -rw
```

注意：

(1) PTE 中输出的权限是 PTE 表中的数据给出的，并没有和 PDE 表中权限做与运算。

(2) 整个 print_pgdir 函数强调两点：第一是相同权限，第二是连续。

(3) print_pgdir 中用到了 vpt 和 vpd 两个变量。可以参考 VPT 和 PGADDR 两个宏。

自映射机制方便用户态程序访问页表。因为页表是内核维护的，用户程序很难知道自己页表的映射结构。VPT 实际上在内核地址空间，我们可以用同样的方式实现一个用户地址空间的映射（比如 pgdir[UVPT]=PADDR(pgdir)|PTE_P|PTE_U，注意，这里不能给写权限，并且 pgdir 是每个进程的 page table，不是 boot_pgdir），这样，用户程序就可以用和内核一样的 print_pgdir 函数遍历自己的页表结构。

3.4 实验报告要求

从网站上下载 lab2.zip 后，解压得到本文档和代码目录 lab2，完成实验中的各个练习。完成代码编写并检查无误后，在对应目录下执行 make handin 任务，即会自动生成 lab2-handin.tar.gz。最后请一定提前或按时提交到网络学堂上。

注意有 lab2 的注释，代码中所有需要完成的地方（Challenge 除外）都有 lab2 和“Your Code”的注释，请在提交时特别注意保持注释，并将“Your Code”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

辅助材料 A 探测物理内存分布和大小的方法

操作系统需要知道整个计算机系统中的物理内存是如何分布的，哪些可用，哪些不可用。其基本方法是通过 BIOS 中断调用来帮助完成的。BIOS 中断调用必须在实模式下进行，所以在 bootloader 进入保护模式前完成这部分工作相对比较合适。这些部分由 boot/bootasm.S 中从 probe_memory 处到 finish_probe 处的代码部分完成。通过 BIOS 中断获取内存可调用参数为 e820h 的 INT 15h BIOS 中断。BIOS 通过系统内存映射地址描述符（Address Range Descriptor）格式来表示系统物理内存布局，其具体表示如下所示。

```
Offset Size Description
00h 8B base address              #系统内存块基地址
08h 8B length in bytes           #系统内存大小
10h 4B type of address range     #内存类型
```

看下面的内容：

```
Values for System Memory Map address type:
01h      memory, available to OS
02h      reserved, not available (e.g. system ROM, memory-mapped device)
03h      ACPI Reclaim Memory (usable by OS after reading ACPI tables)
04h      ACPI NVS Memory (OS is required to save this memory between NVS sessions)
```

```
other      not defined yet--treat as Reserved
```

INT15h BIOS 中断的详细调用参数如下所示。

```
eax: e820h:INT 15的中断调用参数。
edx: 534D4150h (即 4个 ASCII字符"SMAP"),这只是一个签名而已。
ebx: 如果是第一次调用或内存区域扫描完毕,则为 0。如果不是,则存放上次调用之后的计数值。
ecx: 保存地址范围描述符的内存大小,应该大于等于 20B。
es:di: 指向保存地址范围描述符结构的缓冲区,BIOS把信息写入这个结构的起始地址。
```

此中断的返回值如下所示。

```
cflags的 CF位:若 INT 15中断执行成功,则不置位,否则置位。
eax: 534D4150h ('SMAP')。
es:di: 指向保存地址范围描述符的缓冲区,此时缓冲区内的数据已由 BIOS填写完毕。
ebx: 下一个地址范围描述符的计数地址。
ecx: 返回 BIOS往 ES:DI处写的地址范围描述符的字节大小。
ah: 失败时保存出错代码。
```

这样,通过调用 INT 15h BIOS 中断,递增 di 的值(20 的倍数),让 BIOS 帮我们查找出一个一个的内存布局 entry,并放入一个保存地址范围描述符结构的缓冲区中,供后续的 ucore 进一步进行物理内存管理。这个缓冲区结构定义在 memlayout.h 中:

```
struct e820map {
    int nr_map;
    struct {
        long long addr;
        long long size;
        long type;
    } map[E820MAX];
};
```

辅助材料 B　实现物理内存探测

物理内存探测是在 bootasm.S 中实现的,相关代码很短,如下所示。

```
probe_memory:
//对 0x8000处的 32位单元清零,即给位于 0x8000处的
//struct e820map的成员变量 nr_map清零
    movl$ 0, 0x8000
        xorl %ebx, %ebx
//表示设置调用 INT 15h BIOS中断后,BIOS返回的映射地址描述符的起始地址
        movw$ 0x8004, %di
start_probe:
        movl$ 0xE820, %eax // INT 15的中断调用参数
//设置地址范围描述符的大小为 20B,其大小等于 struct e820map的成员变量 map的大小
        movl$ 20, %ecx
```

```
//设置 edx 为 534D4150h (即 4 个 ASCII 字符"SMAP"),这是一个约定
        movl$ SMAP, % edx
//调用 int 0x15 中断,要求 BIOS 返回一个用地址范围描述符表示的内存段信息
        int$ 0x15
//如果 eflags 的 CF 位为 0,则表示还有内存段需要探测
        jnc cont
//探测有问题,结束探测
        movw$ 12345, 0x8000
        jmp finish_probe
cont:
//设置下一个 BIOS 返回的映射地址描述符的起始地址
        addw$ 20, % di
//递增 struct e820map 的成员变量 nr_map
        incl 0x8000
//如果 INT0x15 返回的 ebx 为零,表示探测结束,否则继续探测
        cmpl$ 0, % ebx
        jnz start_probe
finish_probe:
```

上述代码正常执行完毕后,在 0x8000 地址处保存了从 BIOS 中获得的内存分布信息,此信息按照 struct e820map 的设置来进行填充。这部分信息将在 bootloader 启动 ucore 后,由 ucore 的 page_init 函数来根据 struct e820map 的 memmap(定义了起始地址为 0x8000)来完成对整个机器中的物理内存的总体管理。

辅助材料 C　链接地址、虚拟地址、物理地址、加载地址以及 edata/end/text 的含义

1. 链接脚本简介

ucore kernel 的各个部分由组成 kernel 的各个.o 或.a 文件构成,且各个部分在内存中地址位置由 ld 工具根据 kernel.ld 链接脚本(Linker Script)来设定。ld 工具使用命令-T 指定链接脚本。链接脚本主要用于规定如何把输入文件(各个.o 或.a 文件)内的 section 放入输出文件(lab2/bin/kernel,即 ELF 格式的 ucore 内核)内,并控制输出文件内各部分在程序地址空间内的布局。下面简单分析一下/lab2/tools/kernel.ld,来了解一下 ucore 内核的地址布局情况。kernel.ld 的内容如下所示。

```
/* Simple linker script for the ucore kernel.
   See the GNU ld 'info' manual ("info ld") to learn the syntax. */

OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(kern_entry)

SECTIONS {
    /* Load the kernel at this address: "." means the current address */
```

```
.= 0xC0100000;

.text : {
    * (.text .stub .text.* .gnu.linkonce.t.* )
}

PROVIDE(etext= .); /* Define the 'etext' symbol to this value */

.rodata : {
    * (.rodata .rodata.* .gnu.linkonce.r.* )
}

/* Include debugging information in kernel memory */
.stab : {
    PROVIDE(__STAB_BEGIN__ = .);
    * (.stab);
    PROVIDE(__STAB_END__ = .);
    BYTE(0)      /* Force the linker to allocate space
                 for this section */
}

.stabstr : {
    PROVIDE(__STABSTR_BEGIN__ = .);
    * (.stabstr);
    PROVIDE(__STABSTR_END__ = .);
    BYTE(0)      /* Force the linker to allocate space
                 for this section */
}

/* Adjust the address for the data segment to the next page */
.= ALIGN(0x1000);

/* The data segment */
.data : {
    * (.data)
}

PROVIDE(edata= .);

.bss : {
    * (.bss)
}

PROVIDE(end= .);
```

```
    /DISCARD/ : {
        * (.eh_frame .note.GNU-stack)
    }
}
```

其实从链接脚本的内容可以大致猜出它指定告诉链接器的各种信息。

(1) 内核加载地址：0xC0100000。

(2) 入口(起始代码)地址：ENTRY(kern_entry)。

(3) CPU 机器类型：i386。

其最主要的信息是告诉链接器各输入文件的各 section 应该怎么组合：应该从哪个地址开始放，各个 section 以什么顺序放，分别怎么对齐等，最终组成输出文件的各 section。除此之外，linker script 还可以定义各种符号(如.text、.data、.bss 等)，形成最终生成的一堆符号的列表(符号表)，每个符号包含了符号名字、符号所引用的内存地址，以及其他一些属性信息。符号实际上就是一个地址的符号表示，其本身不占用的程序运行的内存空间。

2. 链接地址、加载地址、虚拟地址、物理地址

ucore 设定了 ucore 运行中的虚地址空间，具体设置可看 lab2/kern/mm/memlayout.h 中描述的“Virtual memory map”图，可以了解虚地址和物理地址的对应关系。lab2/tools/kernel.ld 描述的是执行代码的链接地址(link_addr)，比如内核起始地址是 0xC0100000，这是一个虚地址。所以我们可以认为链接地址等于虚地址。在 ucore 建立内核页表时，设定了物理地址和虚地址的虚实映射关系是：

```
Physical Address+ 0xC0000000= Virtual address
```

即虚地址和物理地址之间有一个偏移。但 boot loader 把 ucore kernel 加载到内存时，采用的是加载地址(Load Address)，这是由于 ucore 还没有运行，即还没有启动页表映射，导致这时采用的寻址方式是段寻址方式，用的是 boot loader 在初始化阶段设置的段映射关系，其映射关系(可参看 bootasm.S 的末尾处有关段描述符表的内容)如下：

```
Linear Address= Physical Address= Virtual Address
```

查看 bootloader 的实现代码 bootmain::bootmain.c：

```
readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
```

这里的 ph－>p_va＝0xC0XXXXXX，就是 ld 工具根据 kernel.ld 设置的链接地址，且链接地址等于虚拟地址。考虑到 ph－>p_va & 0xFFFFFF＝＝0x0XXXXXX，所以 bootloader 加载 ucore kernel 的加载地址是 0x0XXXXXX，这实际上是 ucore 内核所在的物理地址。简言之，OS 的链接地址(Link Address)在 tools/kernel.ld 中设置好了，是一个虚地址(Virtual Address)；而 ucore kernel 的加载地址(Load Address)在 boot loader 中的 bootmain 函数中指定，是一个物理地址。

总结一下，ucore 内核的链接地址＝＝ucore 内核的虚拟地址；boot loader 加载 ucore 内核用到的加载地址＝＝ucore 内核的物理地址。

3. edata、end、text 的含义

在基于 ELF 执行文件格式的代码中，存在一些对代码和数据的表述，基本概念如下。

(1) BSS 段(BSS Segment):指用来存放程序中未初始化的全局变量的内存区域。BSS 是英文 Block Started by Symbol 的简称。BSS 段属于静态内存分配。

(2) 数据段(Data Segment):指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。

(3) 代码段(Code Segment/Text Segment):指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定,并且内存区域通常属于只读,某些架构也允许代码段为可写,即允许修改程序。在代码段中,也有可能包含一些只读的常数变量,例如字符串常量等。

在 lab2/kern/init/init.c 的 kern_init 函数中,声明了外部全局变量:

```
extern char edata[],end[];
```

但搜寻所有源码文件 *.[ch],没有发现有这两个变量的定义。那这两个变量从哪里来的呢?其实在 lab2/tools/kernel.ld 中,可以看到如下内容:

```
⋮
.text : {
        * (.text .stub .text.* .gnu.linkonce.t.* )
}
⋮
    .data : {
        * (.data)
}
⋮
    PROVIDE(edata= .);
⋮
    .bss : {
        * (.bss)
}
⋮
    PROVIDE(end= .);
⋮
```

这里的“.”表示当前地址,.text 表示代码段起始地址,.data 也是一个地址,可以看出,它不仅代表了代码段的结束地址,也是数据段的起始地址。以此类推,edata 表示数据段的结束地址,.bss 表示数据段的结束地址和 BSS 段的起始地址,end 表示 BSS 段的结束地址。

这样回头看 kerne_init 中的外部全局变量,可知 edata[]和 end[]这些变量是 ld 根据 kernel.ld 链接脚本生成的全局变量,表示相应段的起始地址或结束地址等,它们不在任何一个.S、.c 或.h 文件中定义。

第 4 章　实验 3：虚拟内存管理

4.1　实 验 目 的

（1）了解虚拟内存的 Page Fault 异常处理实现。

（2）了解页替换算法在操作系统中的实现。

4.2　实 验 内 容

做完实验 2 后，大家可能了解并掌握了物理内存管理中的连续空间分配算法的具体实现以及如何建立二级页表。本次实验是在实验 2 的基础上，借助于页表机制和实验 1 中涉及的中断异常处理机制，完成 Page Fault 异常处理和 FIFO 页替换算法的实现，结合磁盘提供的缓存空间，从而能够支持虚存管理，提供一个比实际物理内存空间“更大”的虚拟内存空间给系统使用。实际操作系统中的虚拟内存管理设计与实现是相当复杂的，涉及与进程管理系统、文件系统等的交叉访问。如果大家有余力，可以尝试完成扩展练习，实现 extended clock 页替换算法。

4.2.1　练习

练习 0：填写已有实验。

本实验依赖实验 1 和实验 2。请把实验 1 和实验 2 的代码填入本实验中代码中有 lab1、lab2 的注释相应部分。

练习 1：给未被映射的地址映射上物理页（需要编程）。

完成 do_pgfault（mm/vmm.c）函数，给未被映射的地址映射上物理页。设置访问权限的时候需要参考页面所在 VMA 的权限，同时需要注意映射物理页时需要操作内存控制结构所指定的页表，而不是内核的页表。

注意：在 lab2 EXERCISE 1 处填写代码。执行 make qemu 后，如果通过 check_pgfault 函数的测试后，会有“check_pgfault() succeeded!”的输出，表示练习 1 基本正确。

练习 2：补充完成基于 FIFO 的页面替换算法（需要编程）。

完成 vmm.c 中的 do_pgfault 函数，并且在实现 FIFO 算法的 swap_fifo.c 中完成 map_swappable 和 swap_out_vistim 函数。通过对 swap 的测试。

注意：在 lab2 EXERCISE 2 处填写代码。执行 make qemu 后，如果通过 check_swap 函数的测试后，会有“check_swap() succeeded!”的输出，表示练习 2 基本正确。

扩展练习 Challenge：实现识别 dirty bit 的 extended clock 页替换算法（需要编程）。

Challenge 部分不是必做部分，不过做正确最后会酌情加分。需写出有详细的设计、分析和测试的实验报告。完成出色的可获得适当加分（基本实验完成后一周内完成，单独提交）。

4.2.2 项目组成

目录结构图如图 4-1 所示。

```
|--boot
|-- kern
|   |-- driver
|   |   |-- …
|   |   |-- ide.c
|   |   `-- ide.h
|   |-- fs
|   |   |-- fs.h
|   |   |-- swapfs.c
|   |   `-- swapfs.h
|   |-- init
|   |   |-- …
|   |   `-- init.c
|   |-- mm
|   |   |-- default_pmm.c
|   |   |-- default_pmm.h
|   |   |-- memlayout.h
|   |   |-- mmu.h
|   |   |-- pmm.c
|   |   |-- pmm.h
|   |   |-- swap.c
|   |   |-- swap.h
|   |   |-- swap_fifo.c
|   |   |-- swap_fifo.h
|   |   |-- vmm.c
|   |   `-- vmm.h
|   |-- sync
|   `-- trap
|       |-- trap.c
|       `-- …
|-- libs
|       |-- list.h
|       `-- …
`-- tools
```

图 4-1　目录结构图

相对于实验 2，实验 3 主要增加的文件有 ide. c、ide. h、fs. h、swapfs. h、swapfs. c、swap. c、swap. h、swap_fifo. c、swap_fifo. h、vmm. c 和 vmm. h。主要修改的文件有 init. c、default_pmm. c、default_pmm. h、pmm. c 和 pmm. n。除这些文件以外的文件为其他需要用到的重要文件。主要改动如下。

(1) kern/mm/default_pmm. [ch]：实现基于 struct pmm_manager 类框架的 FistFit 物理内存分配参考实现(分配最小单位为页，即 4096B)，相关分配页和释放页等实现会间接被 kmalloc/kfree 等函数使用。

(2) kern/mm/pmm. [ch]：pmm. h 定义物理内存分配类框架 struct pmm_manager。pmm. c 包含了对此物理内存分配类框架的访问，以及与建立、修改、访问页表相关的各种函数实现。在本实验中会用到 kmalloc/kfree 等函数。

(3) libs/list. h：定义了通用双向链表结构以及相关的查找、插入等基本操作，这是建立基于链表方法的物理内存管理(以及其他内核功能)的基础。在 lab0 文档中有相关描述。其他有类似双向链表需求的内核功能模块可直接使用 list. h 中定义的函数。在本实验中会多次用到插入、删除等操作函数。

(4) kern/driver/ide.[ch]：定义和实现了内存页 swap 机制所需的磁盘扇区的读写操作支持；在本实验中会涉及通过 swapfs_* 函数间接使用文件中的函数，故了解即可。

(5) kern/fs/*：定义和实现了内存页 swap 机制所需从磁盘读数据到内存页和写内存数据到磁盘上的函数 swapfs_read/swapfs_write。在本实验中会涉及使用这两个函数。

(6) kern/mm/memlayout.h：修改了 struct Page，增加了两项 pra_* 成员结构，其中 pra_page_link 可以用来建立描述各个页访问情况（比如根据访问先后）的链表。在本实验中会涉及使用这两个成员结构，以及 le2page 等宏。

(7) kern/mm/vmm.[ch]：vmm.h 描述了 mm_struct、vma_struct 等表述可访问的虚存地址访问的一些信息，下面会进一步详细讲解。vmm.c 涉及 mm、vma 结构数据的创建、销毁、查找、插入等函数，这些函数在 check_vma、check_vmm 等中被使用，理解即可。page fault 处理相关的 do_pgfault 函数是本次实验需要设计完成的。

(8) kern/mm/swap.[ch]：定义了实现页替换算法的类框架 struct swap_manager。swap.c 包含了对此页替换算法类框架的初始化、页换入/换出等各种函数实现。重点是要理解何时调用 swap_out 和 swap_in 函数。如何实现在此框架下连接具体的页替换算法？check_swap 函数以及被此函数调用的_fifo_check_swap 函数完成了对本次实验中的练习 2：FIFO 页替换算法基本正确性的检查，可了解，便于知道为何产生错误。

(9) kern/mm/swap_fifo.[ch]：FIFO 页替换算法基于类框架 struct swap_manager 的简化实现，主要被 swap.c 的相关函数调用。重点是_fifo_map_swappable 函数（可用于建立页访问属性和关系，比如访问时间的先后顺序）和_fifo_swap_out_victim 函数（可用于实现挑选出要换出的页），当然换出哪个页需要借助于 fifo_map_swappable 函数建立的某种属性关系，已选出合适的页。

(10) kern/mm/mmu.h：其中定义额也页表项的各种属性位，比如 PTE_P\PET_D\PET_A 等，对于实现扩展实验的 clock 算法会有帮助。

本次实验的主要练习集中在 vmm.c 中的 do_pgfault 函数和 swap_fifo.c 中的_fifo_map_swappable 函数、_fifo_swap_out_victim 函数。

编译并运行代码的命令如下：

```
make
make qemu
```

则可以得到如附录所示的显示内容（仅供参考，不是标准答案输出）。

4.3 虚拟内存管理概述

4.3.1 基本原理概述

什么是虚拟内存？简单地说，它是指程序员或 CPU “需要”和直接“看到”的内存，这其实暗示了两点。

(1) 虚拟内存单元不一定有实际的物理内存单元对应，即实际的物理内存单元可能不存在。

(2) 如果虚拟内存单元对应有实际的物理内存单元，那两者的地址一般是不相等的。通过操作系统的某种内存管理和映射技术可建立虚拟内存与实际的物理内存的对应关系，

使得程序员或 CPU 访问的虚拟内存地址会转换为另外一个物理内存地址。

那么这个“虚拟”的作用或意义在哪里体现呢？在操作系统中，虚拟内存其实包含多个虚拟层次，在不同的层次体现了不同的作用。首先，有了分页机制后，程序员或 CPU 直接“看到”的地址已经不是实际的物理地址了，这已经有一层虚拟化，可简称为内存地址虚拟化。有了内存地址虚拟化，就可以通过设置页表项来限定软件运行时的访问空间，确保软件运行不越界，完成内存访问保护的功能。

通过内存地址虚拟化，可以使得软件在没有访问某虚拟内存地址时不分配具体的物理内存，而只有在实际访问某虚拟内存地址时，操作系统再动态地分配物理内存，建立虚拟内存到物理内存的页映射关系，这种技术属于 lazy load 技术，简称按需分页（Demand Paging）。把不经常访问的数据所占的内存空间临时写到硬盘上，这样可以腾出更多的空闲内存空间给经常访问的数据；当 CPU 访问到不经常访问的数据时，再把这些数据从硬盘读入到内存中，这种技术称为页换入换出（Page Swap In/Out）。这种内存管理技术给了程序员更大的内存“空间”，我们称为内存空间虚拟化。

4.3.2 实验执行流程概述

本次实验主要完成 ucore 内核对虚拟内存的管理工作。其总体设计思路比较简单，即首先完成初始化虚拟内存管理机制，即需要设置好哪些页需要放在物理内存中，哪些页不需要放在物理内存中，而是可被换出到硬盘上，并涉及完善建立页表映射、页错误异常处理操作等函数实现。然后就执行一组访存测试，看看我们建立的页表项是否能够正确完成虚实地址映射，是否正确描述了虚拟内存页在物理内存中还是在硬盘上，是否能够正确把虚拟内存页在物理内存和硬盘之间进行传递，是否正确实现了页面替换算法等。lab3 的总体执行流程如下。

首先是初始化过程。参考 ucore 总控函数 init 的代码，可以看到在调用完成虚拟内存初始化的 vmm_init 函数之前，需要首先调用 pmm_init 函数完成物理内存的管理，这也是 lab2 已经完成的内容。接着是执行中断和异常相关的初始化工作，即调用 pic_init 函数和 idt_init 函数等，这些工作与 lab1 的中断异常初始化工作的内容相同。

调用完 idt_init 函数之后，将进一步调用三个 lab3 中才有的新函数 vmm_init、ide_init 和 swap_init。这三个函数设计了本次实验中的两个练习。第一个函数 vmm_init 是检查练习 1 是否正确实现了。为了表述不在物理内存中的“合法”虚拟页，需要有数据结构来描述这样的页，为此 ucore 建立了 mm_struct 和 vma_struct 数据结构（在 4.3.3 小节中有进一步详细描述），假定我们已经描述好了这样的“合法”虚拟页，当 ucore 访问这些“合法”虚拟页时，会由于没有虚实地址映射而产生页错误异常。如果我们正确实现了练习 1，则 do_pgfault 函数会申请一个空闲物理页，并建好虚实映射关系，从而使得这样的“合法”虚拟页有实际的物理页帧对应。这样练习 1 就算完成了。

ide_init 和 swap_init 是为练习 2 准备的。由于页面置换算法的实现存在对硬盘数据块的读写，所以 ide_init 就是完成对用于页换入换出的硬盘（简称 swap 硬盘）的初始化工作。完成 ide_init 函数后，ucore 就可以对这个 swap 硬盘进行读写操作了。swap_init 函数首先建立 swap_manager，swap_manager 是完成页面替换过程的主要功能模块，其中包含了页面置换算法的实现（具体内容可参考 4.5 节）。然后会进一步调用执行 check_swap 函数在内

核中分配一些页，模拟对这些页的访问，这会产生页错误异常。如果我们正确实现了练习2，就可通过 do_pgfault 来调用 swap_map_swappable 函数来查询这些页的访问情况，并间接调用实现页面置换算法的相关函数，把“不常用”的页换出到磁盘上。

ucore 在实现上述技术时，需要解决三个关键问题。

(1) 当程序运行中访问内存产生 page fault 异常时，如何判定这个引起异常的虚拟地址内存访问是越界、写只读页的“非法地址”访问，还是由于数据被临时换出到磁盘上，或还没有分配内存的“合法地址”访问？

(2) 何时进行请求调页/页换入换出处理？

(3) 如何在现有 ucore 的基础上实现页替换算法？

接下来将进一步分析完成 lab3 主要注意的关键问题和涉及的关键数据结构。

4.3.3 关键数据结构和相关函数分析

对于第一个问题的出现，在于实验 2 中有关内存的数据结构和相关操作都是直接针对实际存在的资源——物理内存空间的管理，没有从一般应用程序对内存的“需求”考虑，即需要有相关的数据结构和操作来体现一般应用程序对虚拟内存的“需求”。一般应用程序对虚拟内存的“需求”与物理内存空间的“供给”没有直接的对应关系，ucore 是通过 page fault 异常处理来间接完成这两者之间的衔接。

page_fault 函数不知道哪些是“合法”的虚拟页，原因是 ucore 还缺少一定的数据结构来描述这种不在物理内存中的“合法”虚拟页。为此 ucore 通过建立 mm_struct 和 vma_struct 数据结构，描述了 ucore 模拟应用程序运行所需的合法内存空间。当访问内存产生 page fault 异常时，可获得访问的内存的方式(读或写)以及具体的虚拟内存地址，这样 ucore 就可以查询此地址，看是否属于 vma_struct 数据结构中描述的合法地址范围中，如果在，则可根据具体情况进行请求调页/页换入换出处理(这就是练习 2 涉及的部分)；如果不在，则报错。mm_struct 和 vma_struct 数据结构结合页表表示虚拟地址空间和物理地址空间的示意图如图 4-2 所示。

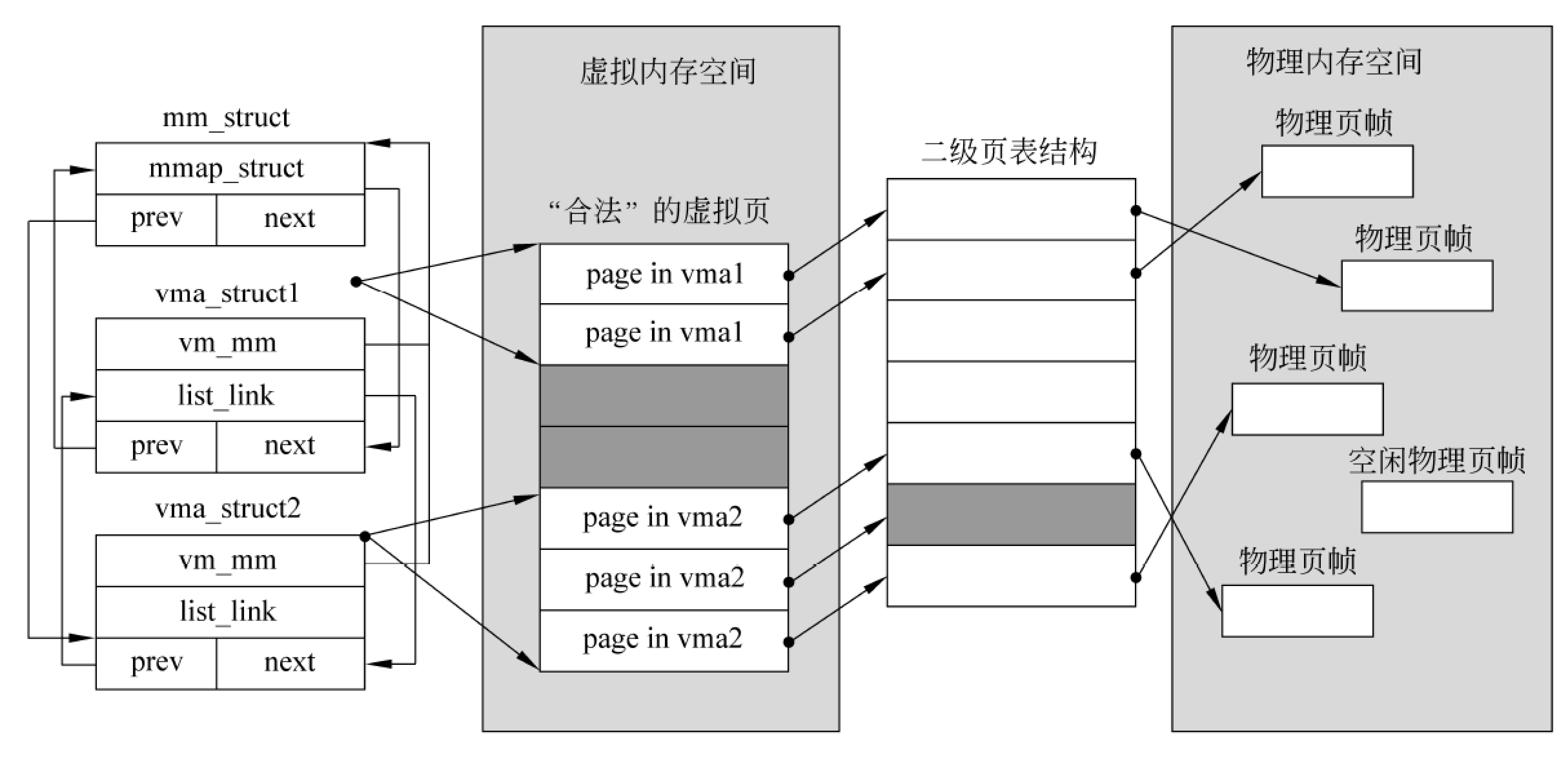


图 4-2 虚拟地址空间和物理地址空间的示意图

在 ucore 中描述应用程序对虚拟内存“需求”的数据结构是 vma_struct（定义在 vmm.h 中），以及针对 vma_struct 的函数操作。这里把一个 vma_struct 结构的变量简称为 vma 变量。vma_struct 的定义如下：

```
struct vma_struct {
    //the set of vma using the same PDT
        struct mm_struct * vm_mm;
        uintptr_t vm_start;                    //start address of vma
        uintptr_t vm_end;                      //end address of vma
        uint32_t vm_flags;                     //flagsess of vma
    //linear list link which sorted by start address of vma
        list_entry_t list_link;
};
```

vm_start 和 vm_end 描述了一个连续地址的虚拟内存空间的起始位置和结束位置，这两个值都应该是 PGSIZE 对齐的，而且描述的是一个合理的地址空间范围（即严格确保 vm_start＜vm_end）；list_link 是一个双向链表，按照从小到大的顺序把一系列用 vma_struct 表示的虚拟内存空间链接起来，并且还要求这些链起来的 vma_struct 应该是不相交的，即 vma 之间的地址空间无交集；vm_flags 表示了这个虚拟内存空间的属性，目前的属性如下：

```
#define VM_READ     0x00000001        //只读
#define VM_WRITE    0x00000002        //可读写
#define VM_EXEC     0x00000004        //可执行
```

vm_mm 是一个指针，指向一个比 vma_struct 更高的抽象层次的数据结构 mm_struct，这里把一个 mm_struct 结构的变量简称为 mm 变量。这个数据结构表示了包含所有虚拟内存空间的共同属性，具体定义如下：

```
struct mm_struct {
    //linear list link which sorted by start address of vma
    list_entry_t mmap_list;
    //current accessed vma, used for speed purpose
    struct vma_struct * mmap_cache;
    pde_t * pgdir;                    //the PDT of these vma
    int map_count;                    //the count of these vma
    void * sm_priv;                   //the private data for swap manager
};
```

mmap_list 是双向链表头，链接了所有属于同一页目录表的虚拟内存空间，mmap_cache 是指向当前正在使用的虚拟内存空间，由于操作系统执行的“局部性”原理，当前正在用到的虚拟内存空间在接下来的操作中可能还会用到，这时就不需要查链表，而是直接使用此指针就可找到下一次要用到的虚拟内存空间。由于 mmap_cache 的引入，可使得 mm_struct 数据结构的查询加速 30% 以上。pgdir 所指向的就是 mm_struct 数据结构所维护的页表。通过访问 pgdir 可以查找某虚拟地址对应的页表项是否存在以及页表项的属性

等。map_count 记录 mmap_list 里面链接的 vma_struct 的个数。sm_priv 指向用来链接记录页访问情况的链表头，这建立了 mm_struct 和后续要讲到的 swap_manager 之间的联系。

涉及 vma_struct 的操作函数也比较简单，主要包括 3 个。

（1）vma_create：创建 vma。

（2）insert_vma_struct：插入一个 vma。

（3）find_vma：查询 vma。

vma_create 函数根据输入参数 vm_start、vm_end、vm_flags 来创建并初始化描述一个虚拟内存空间的 vma_struct 结构变量。insert_vma_struct 函数完成把一个 vma 变量按照其空间位置[vma－＞vm_start，vma－＞vm_end]从小到大的顺序插入所属的 mm 变量中的 mmap_list 双向链表中。find_vma 根据输入参数 addr 和 mm 变量，查找在 mm 变量中的 mmap_list 双向链表中某个 vma 包含此 addr，即 vma－＞vm_start＜＝ addr ＜vma－＞end。这三个函数与后续讲到的 page fault 异常处理有紧密联系。

涉及 mm_struct 的操作函数比较简单，只有 mm_create 和 mm_destroy 两个函数，从字面意思就可以看出是完成 mm_struct 结构的变量创建和删除。在 mm_create 中用 kmalloc 分配了一块空间，所以在 mm_destroy 中也要对应进行释放。在 ucore 运行过程中，会产生描述虚拟内存空间的 vma_struct 结构，所以在 mm_destroy 中也要对这些 mmap_list 中的 vma 进行释放。

4.4　Page Fault 异常处理

对于第 4.3 节提到的第二个关键问题，解决的关键是 page fault 异常处理过程中主要涉及的函数——do_pgfault。在程序的执行过程中由于某种原因（页框不存在/写只读页等）而使 CPU 无法最终访问到相应的物理内存单元，即无法完成从虚拟地址到物理地址的映射时，CPU 会产生一次页错误异常，从而需要进行相应的页错误异常服务例程。这个页错误异常处理的时机就是求调页/页换入换出/处理的执行时机。当相关处理完成后，页错误异常服务例程会返回到产生异常的指令处重新执行，使得软件可以继续正常运行下去。

具体而言，当启动分页机制以后，如果一条指令或数据的虚拟地址所对应的物理页框不在内存中或者访问的类型有错误（比如写一个只读页或用户态程序访问内核态的数据等），就会发生页错误异常。产生页面异常的主要原因如下。

（1）目标页面不存在（页表项全为 0，即该线性地址与物理地址尚未建立映射或者已经撤销）。

（2）相应的物理页面不在内存中（页表项非空，但 Present 标志位＝0，比如在 swap 分区或磁盘文件上），这将在下面介绍换页机制实现时进一步讲解如何处理。

（3）访问权限不符合（此时页表项 P 标志＝1，比如，试图写只读页面）。

当出现上面情况之一，就会产生页面 page fault（＃PF）异常。产生异常的线性地址存储在 CR2 中，并且将是 page fault 的产生类型保存在 error code 中，比如 bit 0 表示是否 PTE_P 为 0，bit 1 表示是否 write 操作。

产生页错误异常后，CPU 硬件和软件都会做一些事情来应对此事。首先页错误异常也

是一种异常,所以针对一般异常的硬件处理操作是必须要做的,即 CPU 在当前内核栈保存当前被打断的程序现场,即依次压入当前被打断程序使用的 Eflags、CS、EIP、Error Code;由于页错误异常的中断号是 0xE,CPU 把异常中断号 0xE 对应的中断异常服务例程的地址(vectors.S 中的标号 vector14 处)加载到 CS 和 EIP 寄存器中,开始执行中断服务例程。这时 ucore 开始处理异常中断,首先需要保存硬件没有保存的寄存器。在 vectors.S 中的标号 vector14 处先把中断号压入内核栈,然后在 trapentry.S 中的标号_alltraps 处把 DS、ES 和其他通用寄存器都压栈。自此,被打断的程序现场被保存在内核栈中。接下来,在 trap.c 的 trap 函数开始了中断服务例程的处理流程,大致调用关系为

```
trap→trap_dispatch→pgfault_handler→do_pgfault
```

下面需要具体分析一下 do_pgfault 函数。do_pgfault 的调用关系如图 4-3 所示。

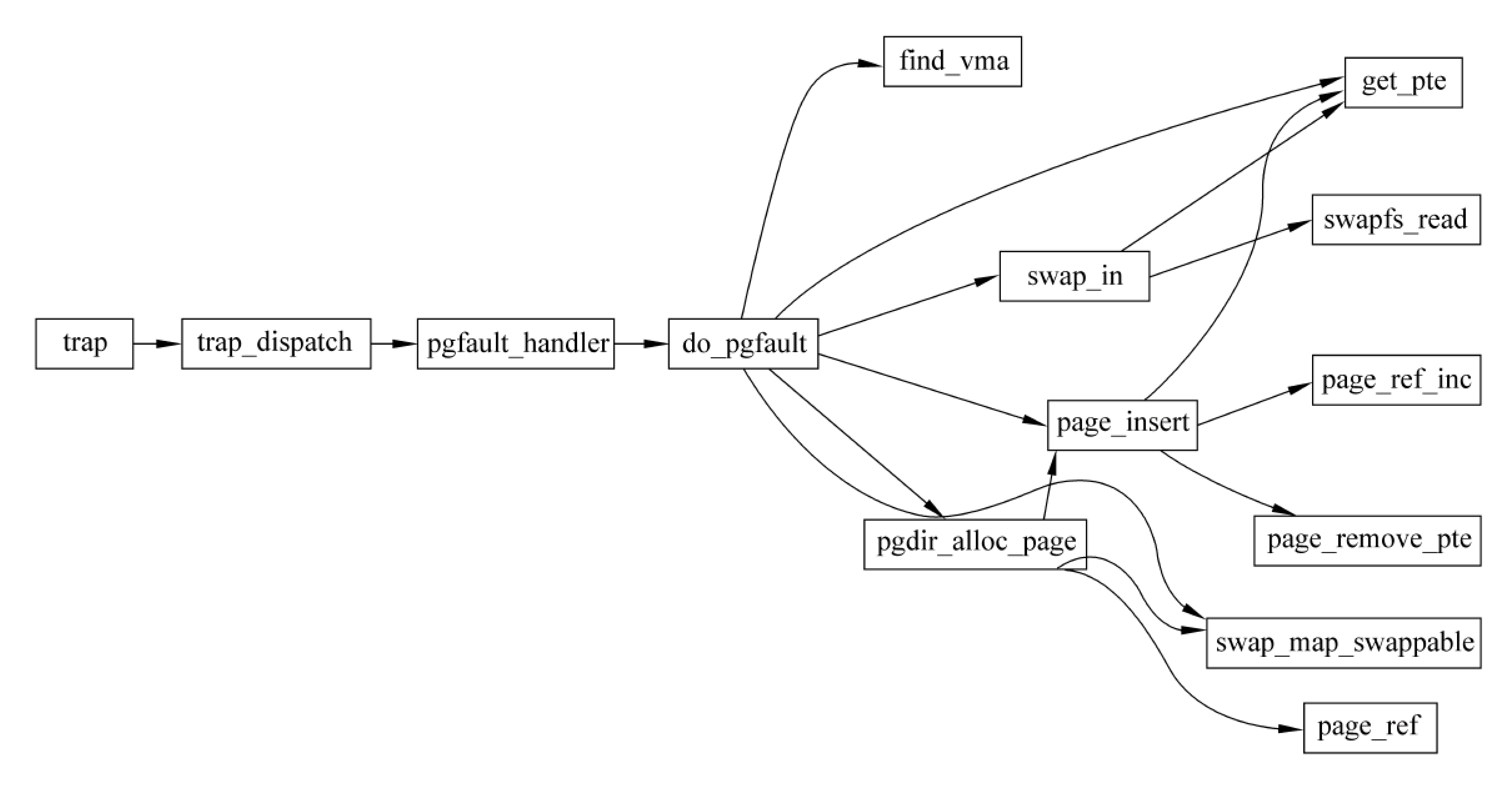


图 4-3　do_pgfault 的调用关系图

产生页错误异常后,CPU 把引起页错误异常的虚拟地址装到寄存器 CR2 中,并给出了出错码(tf－＞tf_err),指示引起页错误异常的存储器访问的类型。而中断服务例程会调用页错误异常处理函数 do_pgfault 进行具体处理。页错误异常处理是实现按需分页、swap in/out 的关键之处。

ucore 中 do_pgfault 函数是完成页错误异常处理的主要函数,它根据从 CPU 的控制寄存器 CR2 中获取的页错误异常的虚拟地址,以及根据 error code 的错误类型来查找此虚拟地址是否在某个 VMA 的地址范围内,并且是否满足正确的读写权限,如果在此范围内并且权限也正确,就认为这是一次合法访问,但没有建立虚实对应关系,所以需要分配一个空闲的内存页,并修改页表完成虚地址到物理地址的映射,刷新 TLB,然后调用 iret 中断,返回到产生页错误异常的指令处重新执行此指令。如果该虚地址不在某 VMA 范围内,这认为是一次非法访问。

4.5 页面置换机制的实现

4.5.1 页替换算法

操作系统为何要进行页面置换呢？这是由于操作系统给用户态的应用程序提供了一个虚拟的“大容量”内存空间，而实际的物理内存空间又没有那么大。所以操作系统就就“瞒着”应用程序，只把应用程序中“常用”的数据和代码放在物理内存中，而不常用的数据和代码放在了硬盘这样的存储介质上。如果应用程序访问的是“常用”的数据和代码，那么操作系统已经放置在内存中了，不会出现什么问题。但当应用程序访问它认为应该在内存中的的数据或代码时，如果这些数据或代码不在内存中，则根据 4.4 节的介绍，会产生页错误异常。这时，操作系统必须能够应对这种页错误异常，即尽快把应用程序当前需要的数据或代码放到内存中，然后重新执行应用程序产生异常的访存指令。如果在把硬盘中对应的数据或代码调入内存前，操作系统发现物理内存已经没有空闲空间了，这时操作系统必须把它认为“不常用”的页换出到磁盘上 ，以腾出内存空闲空间给应用程序所需的数据或代码。

操作系统迟早会碰到没有内存空闲空间而必须要置换出内存中某个“不常用”的页的情况。如何判断内存中哪些是“常用”的页，哪些是“不常用”的页，把“常用”的页保持在内存中，在物理内存空闲空间不够的情况下，把“不常用”的页置换到硬盘上就是页替换算法着重解决的问题。容易理解，一个好的页替换算法会导致页错误异常次数少，这就意味着访问硬盘的次数也少，从而使得应用程序执行的效率高。本次实验涉及的页替换算法(包括扩展练习)如下。

(1) 先进先出(First In First Out, FIFO)页替换算法：该算法总是淘汰最先进入内存的页，即选择在内存中驻留时间最久的页予以淘汰。只需把一个应用程序在执行过程中已调入内存的页按先后次序链接成一个队列，队列头指向内存中驻留时间最久的页，队列尾指向最近被调入内存的页。这样需要淘汰页时，从队列头很容易查找到需要淘汰的页。FIFO 算法只是在应用程序按线性顺序访问地址空间时效果才好，否则效率不高。因为那些常被访问的页，往往在内存中也停留得最久，结果它们因变“老”而不得不被置换出去。FIFO 算法的另一个缺点是，它有一种异常现象(Belady 现象)，即在增加放置页的页帧的情况下，反而使页错误异常次数增多。

(2) 时钟(Clock)页替换算法，也称最近未使用 (Not Used Recently, NUR) 页替换算法。虽然二次机会算法是一个较合理的算法，但它经常需要在链表中移动页面，这样做既降低了效率，又是不必要的。一个更好的办法是把各个页面组织成环形链表的形式，类似于一个钟的表面。然后把一个指针指向最古老的那个页面，或者说，最先进来的那个页面。时钟算法和第二次机会算法的功能是完全一样的，只是在具体实现上有所不同。时钟算法需要在页表项(PTE)中设置了一位访问位来表示此页表项对应的页当前是否被访问过。当该页被访问时，CPU 中的 MMU 硬件将把访问位置 1。然后将内存中所有的页都通过指针链接起来并形成一个循环队列。初始时，设置一个当前指针指向某页(比如最古老的那个页面)。操作系统需要淘汰页时，对当前指针指向的页所对应的页表项进行查询，如果访问位为 0，则淘汰该页，把它换出到硬盘上；如果访问位为 1，这将该页表项的此位置 0，继续访问下一

个页。该算法近似地体现了 LRU 的思想，且易于实现，开销少。但该算法需要硬件支持来设置访问位，且该算法在本质上与 FIFO 算法是类似的，唯一不同的是在 Clock 算法中跳过了访问位为 1 的页。

(3) 改进的时钟(Enhanced Clock)页替换算法：在时钟置换算法中，淘汰一个页面时只考虑了页面是否被访问过，但在实际情况中，还应考虑被淘汰的页面是否被修改过。因为淘汰修改过的页面还需要写回硬盘，使得其置换代价大于未修改过的页面。改进的时钟置换算法除了考虑页面的访问情况，还需考虑页面的修改情况。即该算法不但希望淘汰的页面是最近未使用的页，而且还希望被淘汰的页是在主存驻留期间其页面内容未被修改过的。这需要为每一页的对应页表项内容中增加一位引用位和一位修改位。当该页被访问时，CPU 中的 MMU 硬件将把访问位置 1。当该页被“写”时，CPU 中的 MMU 硬件将把修改位置 1。这样这两位就存在四种可能的组合情况：(0,0)表示最近未被引用也未被修改，首先选择此页淘汰；(0,1)最近未被使用，但被修改，其次选择；(1,0)最近使用而未修改，再次选择；(1,1)最近使用且修改，最后选择。该算法与时钟算法相比，可进一步减少磁盘的 I/O 操作次数，但为了查找到一个尽可能适合淘汰的页面，可能需要经过多次扫描，这增加了算法本身的执行开销。

4.5.2 页面置换机制

如果要实现页面置换机制，只考虑页替换算法的设计与实现是远远不够的，还需考虑其他问题。

(1) 哪些页可以被换出？

(2) 一个虚拟的页如何与硬盘上的扇区建立对应关系？

(3) 何时进行换入和换出操作？

(4) 如何设计数据结构已支持页替换算法？

(5) 如何完成页的换入和换出操作？

这些问题在下面会逐一进行分析。注意，在实验 3 中仅实现了简单的页面置换机制，但现在还没有涉及实验 4 和实验 5 才实现的内核线程和用户进程，所以还无法通过内核线程机制实现一个完整意义上的虚拟内存页面置换功能。

1. 可以被换出的页

在操作系统的设计中，一个基本的原则是：并非所有的物理页都可以交换出去，只有映射到用户空间且被用户程序直接访问的页面才能被交换，而被内核直接使用的内核空间的页面不能被换出。这里面的原因是什么呢？操作系统是执行的关键代码，需要保证运行的高效性和实时性，如果在操作系统执行过程中，发生了缺页现象，则操作系统不得不等很长时间(硬盘的访问速度比内存的访问速度慢 2～3 个数量级)，这将导致整个系统运行低效，而且，不难想象，处理缺页过程所用到的内核代码或者数据如果被换出，整个内核都面临崩溃的危险。

但在实验 3 实现的 ucore 中，我们只是实现了换入和换出机制，还没有设计用户态执行的程序，所以在实验 3 中仅仅通过执行 check_swap 函数在内核中分配一些页，模拟对这些页的访问，然后通过 do_pgfault 来调用 swap_map_swappable 函数来查询这些页的访问情况并间接调用相关函数，换出“不常用”的页到磁盘上。

2. 虚存中的页与硬盘上的扇区之间的映射关系

如果一个页被置换到了硬盘上，那么操作系统如何能简捷地表示这种情况呢？在 ucore 的设计上，充分利用了页表中的 PTE 来表示这种情况：当一个 PTE 用来描述一般意义上的物理页时，显然它应该维护各种权限和映射关系，以及应该有 PTE_P 标记；但当它用来描述一个被置换出去的物理页时，它被用来维护该物理页与 swap 磁盘上扇区的映射关系，并且该 PTE 不应该由 MMU 将它解释成物理页映射(即没有 PTE_P 标记)，与此同时，对应的权限则交由 mm_struct 来维护，当对位于该页的内存地址进行访问的时候，必然导致 page fault，然后 ucore 能够根据 PTE 描述的 swap 项将相应的物理页重新建立起来，并根据虚存所描述的权限重新设置好 PTE 使得内存访问能够继续正常进行。

如果一个页(4KB/页)被置换到了硬盘某 8 个扇区(0.5KB/扇区)，该 PTE 的最低位——present 位应该为 0（即 PTE_P 标记为空，表示虚实地址映射关系不存在)，接下来的 7 位暂时保留，可以用做各种扩展；而原来表示页帧号的高 24 位地址，恰好可以用来表示此页在硬盘上的起始扇区的位置(其从第几个扇区开始)。为了在页表项中区别 0 和 swap 分区的映射，将 swap 分区的一个 page 空出来不用，也就是说一个高 24 位不为 0，而最低位为 0 的 PTE 表示了一个放在硬盘上的页的起始扇区号(见 swap.h 中对 swap_entry_t 的描述，如图 4-4 所示)：

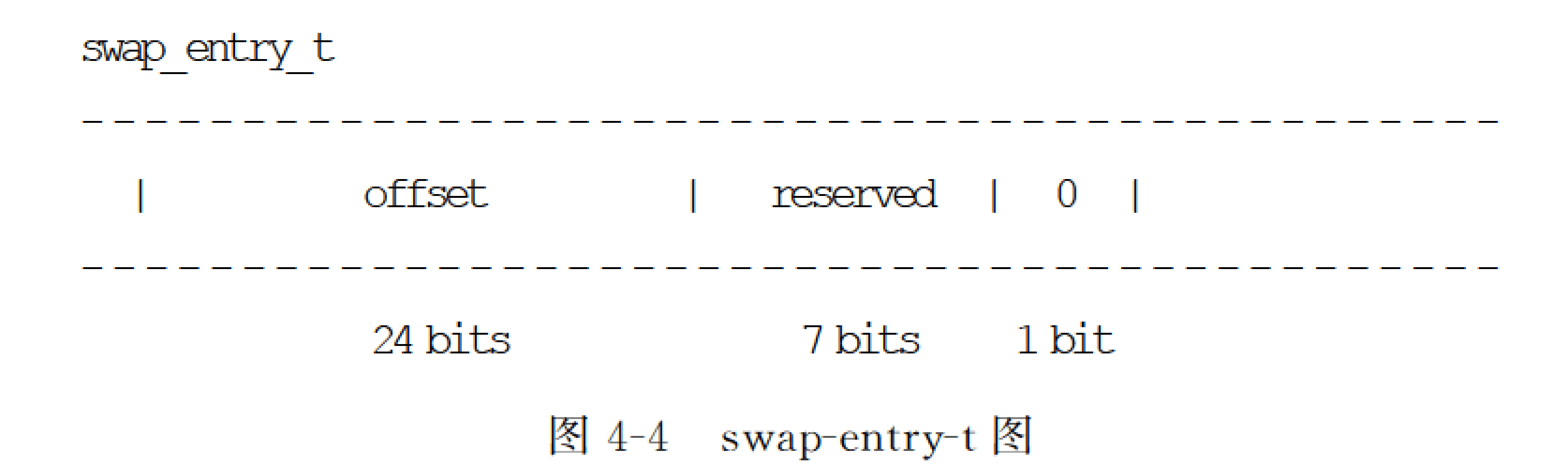


图 4-4　swap-entry-t 图

考虑到硬盘的最小访问单位是一个扇区，而一个扇区的大小为 512(2^8)B，所以需要 8 个连续扇区才能放置一个 4KB 的页。在 ucore 中，用了第二个 IDE 硬盘来保存被换出的扇区，根据实验 3 的输出信息

“ide 1: 262144(sectors), 'QEMU HARDDISK'.”

我们可以知道实验 3 能够保存 262144/8＝32768 个页，即 128MB 的内存空间。swap 分区的大小是 swapfs_init 里面根据磁盘驱动的接口计算出来的，目前 ucore 里面要求 swap 磁盘至少包含 1000 个 page，并且至多能使用 2^{24} 个 page。

3. 执行换入和换出的时机

在实验 3 中，check_mm_struct 变量这个数据结构表示了目前 ucore 认为合法的所有虚拟内存空间集合，而 mm 中的每个 vma 表示了一段地址连续的合法虚拟空间。当 ucore 或应用程序访问地址所在的页不在内存时，就会产生 page fault 异常，引起调用 do_pgfault 函数，此函数会判断产生访问异常的地址属于 check_mm_struct 某个 vma 表示的合法虚拟地址空间，且保存在硬盘 swap 文件中(即对应的 PTE 的高 24 位不为 0，而最低位为 0)，则是执行页换入的时机，将调用 swap_in 函数完成页面换入。

换出页面的时机相对复杂一些，针对不同的策略有不同的时机。ucore 目前大致有两种策略，即积极换出策略和消极换出策略。积极换出策略是指操作系统周期性地(或在系统

不忙的时候)主动把某些认为“不常用”的页换出到硬盘上,从而确保系统中总有一定数量的空闲页存在,这样当需要空闲页时,基本上能够及时满足需求。消极换出策略是指,只是当试图得到空闲页时,发现当前没有空闲的物理页可供分配,这时才开始查找“不常用”页面,并把一个或多个这样的页换出到硬盘上。

在实验 3 中的基本练习中,支持上述第二种情况。对于第一种积极换出策略,即每隔 1s 执行一次的实现积极的换出策略,可考虑在扩展练习中实现。对于第二种消极的换出策略,则是在 ucore 调用 alloc_pages 函数获取空闲页时,此函数如果发现无法从物理内存页分配器(比如 First Fit)获得空闲页,就会进一步调用 swap_out 函数换出某页,实现一种消极的换出策略。

4. 页替换算法的数据结构设计

到实验 2 为止,我们知道目前表示内存中物理页使用情况的变量是基于数据结构 Page 的全局变量 pages 数组,pages 的每一项表示了计算机系统中一个物理页的使用情况。为了表示物理页可被换出或已被换出的情况,可对 Page 数据结构进行扩展:

```
struct Page {
⋮
list_entry_t        pra_page_link;
uintptr_t           pra_vaddr;
};
```

pra_page_link 可用来构造按页的第一次访问时间进行排序的一个链表,这个链表的开始表示第一次访问时间最近的页,链表结尾表示第一次访问时间最远的页。当然链表头可以就可设置为 pra_list_head(定义在 swap_fifo.c 中),构造的时机是在 page fault 发生后,进行 do_pgfault 函数时。pra_vaddr 可以用来记录此物理页对应的虚拟页起始地址。

当一个物理页(struct Page)需要被 swap 出去的时候,首先需要确保它已经分配了一个位于磁盘上的 swap page(由连续的 8 个扇区组成)。这里为了简化设计,在 swap_check 函数中建立了每个虚拟页唯一对应的 swap page,其对应关系设定为:虚拟页对应的 PTE 的索引值=swap page 的扇区起始位置×8。

为了实现各种页替换算法,下面设计了一个页替换算法的类框架 swap_manager:

```
struct swap_manager
{
    const char * name;
    /* Global initialization for the swap manager * /
    int(* init)   (void);
    /* Initialize the priv data inside mm_struct * /
    int(* init_mm)   (struct mm_struct * mm);
    /* Called when tick interrupt occured * /
    int (* tick_event)   (struct mm_struct * mm);
    /* Called when map a swappable page into the mm_struct * /
    int (* map_swappable)   (struct mm_struct * mm, uintptr_t addr, struct Page * page, int swap_in);
    /* When a page is marked as shared, this routine is called to delete the addr entry from the swap
```

```
    manager* /
    int (* set_unswappable) (struct mm_struct* mm, uintptr_t addr);
    /* Try to swap out a page, return then victim* /
    int(* swap_out_victim) (struct mm_struct* mm, struct Page**ptr_page, int in_tick);
    /* check the page relpacement algorithm* /
    int (* check_swap)(void);
};
```

这里关键的两个函数指针是 map_swappable 和 swap_out_vistim，前一个函数用于记录页访问情况的相关属性，后一个函数用于挑选需要换出的页。显然第二个函数依赖于第一个函数记录的页访问情况。tick_event 函数指针也很重要，结合定时产生的中断，可以实现一种积极的换页策略。

5. swap_check 的检查实现

下面具体讲述一下实验 3 中实现置换算法的页面置换的检查执行逻辑，便于大家实现练习 2。实验 3 页面置换的检查过程在函数 swap_check（kern/mm/swap. c 中）中，其大致流程如下。

（1）调用 mm_create 建立 mm 变量，并调用 vma_create 创建 vma 变量，设置合法的访问范围为 4～24KB。

（2）调用 free_page 等操作，模拟形成一个只有 4 个空闲 physical page；并设置了从 4～24KB 的连续 5 个虚拟页的访问操作。

（3）设置记录缺页次数的变量 pgfault_num＝0，执行 check_content_set 函数，使得起始地址分别对起始地址为 0x1000、0x2000、0x3000、0x4000 的虚拟页按时间顺序先后写操作访问，由于之前没有建立页表，所以会产生 page fault 异常，如果完成练习 1，则这些从 4～20KB的 4 个虚拟页会与 ucore 保存的 4 个物理页帧建立映射关系。

（4）对虚页对应的新产生的页表项进行合法性检查。

（5）进入测试页替换算法的主体，执行函数 check_content_access，并进一步调用到_fifo_check_swap 函数，如果通过了所有的 assert。这进一步表示 FIFO 页替换算法基本正确实现。

（6）恢复 ucore 环境。

4.6 实验报告要求

从网站上下载 lab3. zip 后，解压得到本文档和代码目录 lab3，完成实验中的各个练习。完成代码编写并检查无误后，在对应目录下执行 make handin 任务，即会自动生成 lab3-handin. tar. gz。最后请一定提前或按时提交到网络学堂上。

注意有 lab3 的注释，代码中所有需要完成的地方（Challenge 除外）都有 lab3 和“Your Code”的注释，请在提交时特别注意保持注释，并将“Your Code”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。所有扩展实验的加分总和不超过 10 分。

辅助材料 A：正确输出的参考

```
yuchen@yuchen-PAI4:~/oscourse/2012spring/lab3/lab3-code-2012$ make qemu
(THU.CST) os is loading...

Special kernel symbols:
    entry   0xc010002c (phys)
    etext   0xc010962b (phys)
    edata   0xc0122ac8 (phys)
    end     0xc0123c10 (phys)
Kernel executable memory footprint: 143KB
memory management: default_pmm_manager
e820map:
    memory: 0009f400, [00000000, 0009f3ff], type=1.
    memory: 00000c00, [0009f400, 0009ffff], type=2.
    memory: 00010000, [000f0000, 000fffff], type=2.
    memory: 07efd000, [00100000, 07ffcfff], type=1.
    memory: 00003000, [07ffd000, 07ffffff], type=2.
    memory: 00040000, [fffc0000, ffffffff], type=2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
--------------------- BEGIN ---------------------
PDE(0e0) c0000000-f8000000 38000000 urw
    |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
    |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
    |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ---------------------
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:      262144(sectors), 'QEMU HARDDISK'.
SWAP: manager = fifo swap manager
BEGIN check_swap: count 1, total 31992
    mm->sm_priv c0123c04 in fifo_init_mm
setup Page Table for vaddr 0X1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!
setup init env for check_swap begin!
page fault at 0x00001000: K/W [no page found].
page fault at 0x00002000: K/W [no page found].
```

```
page fault at 0x00003000: K/W [no page found].
page fault at 0x00004000: K/W [no page found].
set up init env for check_swap over!
write Virt Page c in fifo_check_swap
write Virt Page a in fifo_check_swap
write Virt Page d in fifo_check_swap
write Virt Page b in fifo_check_swap
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
write Virt Page b in fifo_check_swap
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
write Virt Page b in fifo_check_swap
page fault at 0x00002000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
write Virt Page c in fifo_check_swap
page fault at 0x00003000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in fifo_check_swap
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
check_swap() succeeded!
++ setup timer interrupts
100 ticks
End of Test.
kernel panic at kern/trap/trap.c:20:
    EOT: kernel seems ok.
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
```

第 5 章　实验 4：内核线程管理

5.1　实验目的

（1）了解内核线程创建/执行的管理过程。

（2）了解内核线程的切换和基本调度过程。

5.2　实验内容

实验 2 和实验 3 完成了物理和虚拟内存管理，这给创建内核线程（内核线程是一种特殊的进程）打下了提供内存管理的基础。当一个程序加载到内存中运行时，首先通过 ucore 的内存管理分配合适的空间，然后就需要考虑如何使用 CPU 来“并发”执行多个程序。

本次实验将首先接触的是内核线程的管理。内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：内核线程只运行在内核态而用户进程会在在用户态和内核态交替运行；所有内核线程直接使用共同的 ucore 内核内存空间，不需为每个内核线程维护单独的内存空间，而用户进程需要维护各自的用户内存空间。相关原理介绍可看本章附录 B。

5.2.1　练习

练习 0：填写已有实验。

本实验依赖实验 1～实验 3。请把已做的实验 1～实验 3 的代码填入本实验中代码中有 lab1、lab2、lab3 的注释相应部分。

练习 1：分配并初始化一个进程控制块（需要编码）。

alloc_proc 函数（位于 kern/process/proc.c 中）负责分配并返回一个新的 struct proc_struct 结构，用于存储新建立的内核线程的管理信息。ucore 需要对这个结构进行最基本的初始化，本练习要求完成这个初始化过程。

提示：*在 alloc_proc 函数的实现中，需要初始化的 proc_struct 结构中的成员变量至少包括 state/pid/runs/kstack/need_resched/parent/mm/context/tf/cr3/flags/name。*

练习 2：为新创建的内核线程分配资源（需要编码）。

创建一个内核线程需要分配和设置好很多资源。kernel_thread 函数通过调用 do_fork 函数完成具体内核线程的创建工作。do_kernel 函数会调用 alloc_proc 函数来分配并初始化一个进程控制块，但 alloc_proc 只是找到了一小块内存用以记录进程的必要信息，并没有实际分配这些资源。ucore 一般通过 do_fork 实际创建新的内核线程。do_fork 的作用是，创建当前内核线程的一个副本，它们的执行上下文、代码、数据都一样，但是存储位置不同。在这个过程中，需要给新内核线程分配资源，并且复制原进程的状态。需要完成在 kern/

process/proc. c 中的 do_fork 函数中的处理过程。它的大致执行步骤如下。

(1) 调用 alloc_proc,首先获得一块用户信息块。

(2) 为进程分配一个内核栈。

(3) 复制原进程的内存管理信息到新进程(但内核线程不必做此事)。

(4) 复制原进程上下文到新进程。

(5) 将新进程添加到进程列表。

(6) 唤醒新进程。

(7) 返回新进程号。

练习 3:阅读代码,理解 proc_run 和它调用的函数如何完成进程切换的(无编码工作)。

完成代码编写后,编译并运行代码:

```
make qemu
```

如果可以得到如本章附录 A 所示的显示内容(仅供参考,不是标准答案输出),则基本正确。

扩展练习 Challenge:实现支持任意大小的内存分配算法。

这不是本实验的内容,其实是上一次实验内存的扩展,但考虑到现在的 slab 算法比较复杂,有必要实现一个比较简单的任意大小内存分配算法。可参考本实验中的 slab 如何调用基于页的内存分配算法(注意:不需要关注 slab 的具体实现)来实现 first-fit/best-fit/worst-fit/buddy 等支持任意大小的内存分配算法。

注意:下面是相关的 Linux 实现文档,可供参考。

slob:

http://en. wikipedia. org/wiki/SLOB 和 http://lwn. net/Articles/157944/

slab:

https://www. ibm. com/developerworks/cn/linux/l-linux-slab-allocator/

5.2.2 项目组成

目录结构图如图 5-1 所示。

相对于实验 3,实验 4 主要增加的文件有 rb_tree. c、rb_tree. h、kmalloc. c、kmalloc. h、hash. c 和 unistd. h 等文件。主要修改的文件有 init. c、memlayout. h、pmm. c、pmm. h、swap. c 和 vmm. c,主要改动如下。

(1) kern/process/(新增进程管理相关文件)。

proc. [ch]:新增,实现进程、线程相关功能,包括创建进程/线程,初始化进程/线程,处理进程/线程退出等功能。

entry. S:新增,内核线程入口函数 kernel_thread_entry 的实现。

switch. S:新增,上下文切换,利用堆栈保存、恢复进程上下文 。

(2) kern/init/。

init. c:修改,完成进程系统初始化,并在内核初始化后切入 idle 进程。

(3) kern/mm/(基本上与本次实验没有太直接的联系,了解 kmalloc 和 kfree 如何使用即可)。

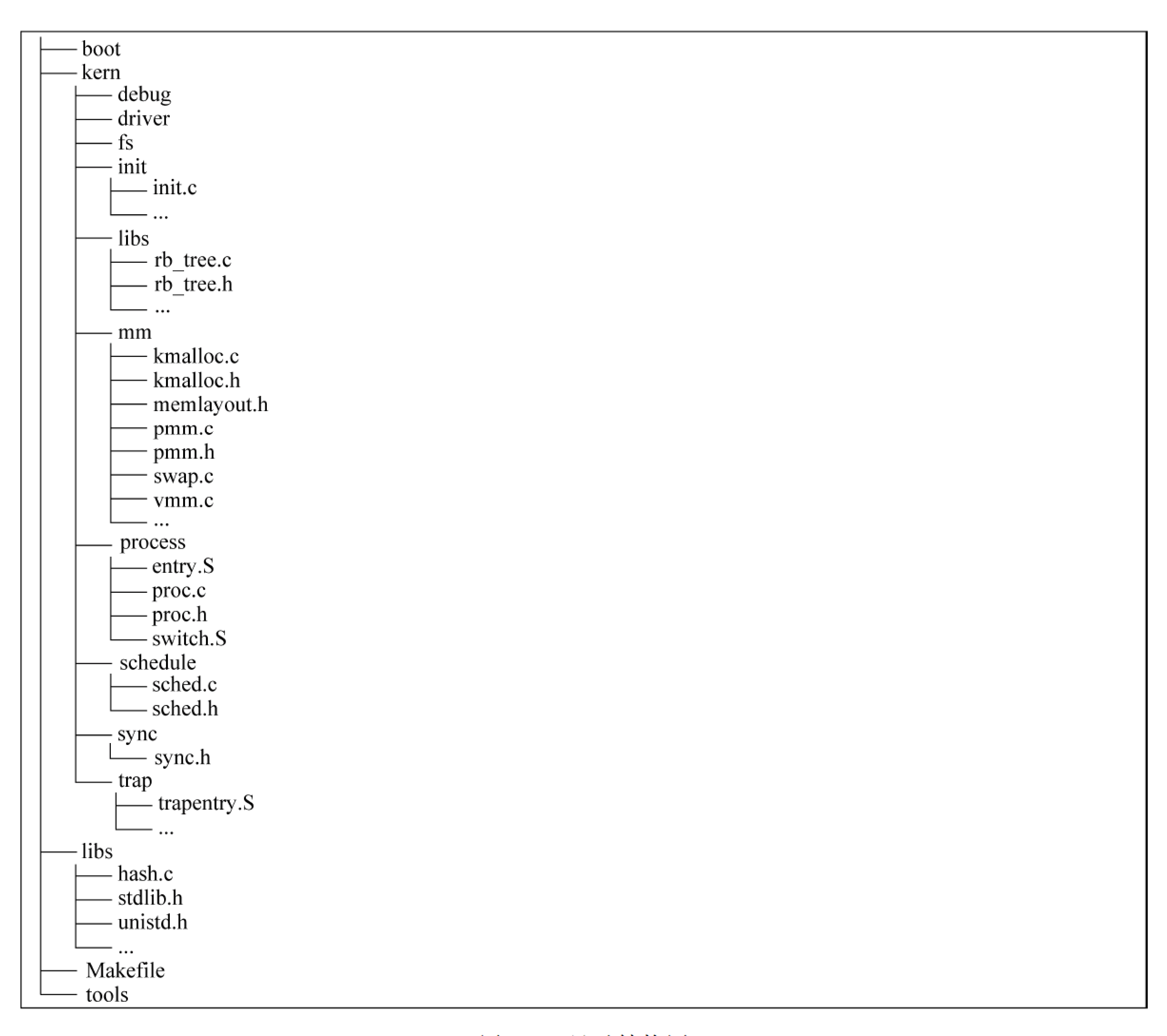


图 5-1　目录结构图

kmalloc.[ch]：新增，定义和实现了新的 kmalloc/kfree 函数。具体实现是基于 slab 分配的简化算法（只要求会调用这两个函数即可）。

memlayout.h：增加 slab 物理内存分配相关的定义与宏（可不用理会）。

pmm.[ch]：修改，在 pmm.c 中添加了调用 kmalloc_init 函数，取消了旧的 kmalloc/kfree 的实现；在 pmm.h 中取消了旧的 kmalloc/kfree 的定义。

swap.c：修改，取消了用于 check 的 Line 185 的执行。

vmm.c：修改，调用新的 kmalloc/kfree。

（4）kern/trap/。

trapentry.S：增加了汇编写的函数 forkrets，用于 do_fork 调用的返回处理。

（5）kern/schedule/。

sched.[ch]：新增，实现 FIFO 策略的进程调度。

（6）kern/libs。

rb_tree.[ch]：新增，实现红黑树，被 slab 分配的简化算法使用（可不用理会）。

编译并运行代码的命令如下：

```
make
make qemu
```

则可以得到如本章附录 A 所示的显示内容(仅供参考,不是标准答案输出)。

5.3 内核线程管理

5.3.1 实验执行流程概述

lab2 和 lab3 完成了对内存的虚拟化,但整个控制流还是一条线串行执行。lab4 将在此基础上进行 CPU 的虚拟化,即让 ucore 实现分时共享 CPU,实现多条控制流能够并发执行。在某种程度上,可以把控制流看做一个内核线程。本次实验将首先接触的是内核线程的管理。内核线程是一种特殊的进程,内核线程与用户进程的区别有两个:内核线程只运行在内核态而用户进程会在用户态和内核态交替运行;所有内核线程直接使用共同的 ucore 内核内存空间,不需为每个内核线程维护单独的内存空间,而用户进程需要维护各自的用户内存空间。从内存空间占用情况这个角度上看,可以把线程看做一种共享内存空间的轻量级进程。

为了实现内核线程,需要设计管理线程的数据结构,即进程控制块(在这里也可叫做线程控制块)。如果要让内核线程运行,首先要创建内核线程对应的进程控制块,还需把这些进程控制块通过链表连在一起,便于随时进行插入、删除和查找操作等进程管理事务。这个链表就是进程控制块链表。然后再通过调度器(Scheduler)来让不同的内核线程在不同的时间段占用 CPU 执行,实现对 CPU 的分时共享。那么 lab4 中是如何一步一步实现这个过程的呢?

还是从 lab4/kern/init/init.c 中的 kern_init 函数入手分析。在 kern_init 函数中,当完成虚拟内存的初始化工作后,就调用了 proc_init 函数,这个函数完成了 idleproc 内核线程和 initproc 内核线程的创建或复制工作,这也是本次实验要完成的练习。idleproc 内核线程的工作就是不停地查询,看是否有其他内核线程可以执行了,如果有,马上让调度器选择那个内核线程执行(请参考 cpu_idle 函数的实现)。所以 idleproc 内核线程是在 ucore 操作系统没有其他内核线程可执行的情况下才会被调用。接着就是调用 kernel_thread 函数来创建 initproc 内核线程。initproc 内核线程的工作就是显示"Hello World",表明自己存在且能正常工作了。

调度器会在特定的调度点上执行调度,完成进程切换。在 lab4 中,这个调度点只有一处,即在 cpu_idle 函数中,此函数如果发现当前进程(也就是 idleproc)的 need_resched 置为 1(在初始化 idleproc 的进程控制块时就置为 1 了),则调用 schedule 函数,完成进程调度和进程切换。进程调度的过程其实比较简单,就是在进程控制块链表中查找到一个"合适"的内核线程,所谓"合适"就是指内核线程处于 PROC_RUNNABLE 状态。在接下来的 switch_to 函数(在后续有详细分析,有一定难度,需深入了解一下)完成具体的进程切换过程。一旦切换成功,那么 initproc 内核线程就可以通过显示字符串来表明本次实验成功。

接下来将主要介绍进程创建所需的重要数据结构——进程控制块 proc_struct,以及

ucore 创建并执行内核线程 idleproc 和 initproc 的两种不同方式，特别是创建 initproc 的方式将被延续到实验 5 中，扩展为创建用户进程的主要方式。另外，还初步涉及了进程调度（实验 6 涉及并会扩展）和进程切换内容。

5.3.2 设计关键数据结构——进程控制块

在实验 4 中，进程管理信息用 struct proc_struct 表示，在 kern/process/proc.h 中定义如下：

```
struct proc_struct {
    enum proc_state state;              //Process state
    int pid;                            //Process ID
    int runs;                           //the running times of Proces
    uintptr_t kstack;                   //Process kernel stack
    volatile bool need_resched;         //need to be rescheduled to release CPU?
    struct proc_struct * parent;        //the parent process
    struct mm_struct * mm;              //Process's memory management field
    struct context context;             //Switch here to run process
    struct trapframe * tf;              //Trap frame for current interrupt
    uintptr_t cr3;                      //the base address of Page Directroy Table(PDT)
    uint32_t flags;                     //Process flag
    char name[PROC_NAME_LEN+1];         //Process name
    list_entry_t list_link;             //Process link list
    list_entry_t hash_link;             //Process hash list
};
```

下面重点解释一下几个比较重要的成员变量。

（1）mm：内存管理的信息，包括内存映射列表、页表指针等。mm 成员变量在 lab3 中用于虚存管理。但在实际 OS 中，内核线程常驻内存，不需要考虑 swap page 问题，在 lab5 中涉及用户进程，才考虑进程用户内存空间的 swap page 问题，mm 才会发挥作用。所以在 lab4 中 mm 对于内核线程就没有用，这样内核线程的 proc_struct 的成员变量 * mm＝0 是合理的。mm 里有个很重要的项 pgdir，记录的是该进程使用的一级页表的物理地址。由于 * mm＝NULL，所以在 proc_struct 数据结构中需要有一个代替 pgdir 项来记录页表起始地址，这就是 proc_struct 数据结构中的 cr3 成员变量。

（2）state：进程所处的状态。

（3）parent ：用户进程的父进程（创建它的进程）。在所有进程中，只有一个进程没有父进程，就是内核创建的第一个内核线程 idleproc。内核根据这个父子关系建立一个树形结构，用于维护一些特殊的操作，例如，确定某个进程是否可以对另外一个进程进行某种操作等。

（4）context：进程的上下文，用于进程切换（参见 switch.S）。在 ucore 中，所有的进程在内核中也是相对独立的（例如，独立的内核堆栈以及上下文等）。使用 context 保存寄存器的目的就在于在内核态中能够进行上下文之间的切换。实际利用 context 进行上下文切换的函数是在 kern/process/switch.S 中定义的 switch_to。

(5) tf：中断帧的指针，总是指向内核栈的某个位置。当进程从用户空间跳转到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。除此之外，ucore 内核允许嵌套中断。因此为了保证嵌套中断发生时 tf 总是能够指向当前的 trapframe，ucore 在内核栈上维护了 tf 的链，可以参考 trap.c::trap 函数做进一步的了解。

(6) cr3：cr3 保存页表的物理地址，目的是进程切换的时候方便直接使用 lcr3 实现页表切换，避免每次都根据 mm 来计算 cr3。mm 数据结构是用来实现用户空间的虚存管理的，但是内核线程没有用户空间，它执行的只是内核中的一小段代码(通常是一小段函数)，所以它没有 mm 结构，也就是 NULL。当某个进程是一个普通用户态进程的时候，PCB 中的 cr3 就是 mm 中页表(pgdir)的物理地址；而当它是内核线程的时候，cr3 等于 boot_cr3。boot_cr3 指向了 ucore 启动时建好的栈内核虚拟空间的页目录表首地址。

(7) kstack：每个线程都有一个内核栈，并且位于内核地址空间的不同位置。对于内核线程，该栈就是运行时的程序使用的栈；而对于普通进程，该栈是发生特权级改变的时候使保存被打断的硬件信息用的栈。ucore 在创建进程时分配了 2 个连续的物理页(参见 memlayout.h 中 KSTACKSIZE 的定义)作为内核栈的空间。这个栈很小，所以内核中的代码应该尽可能地紧凑，并且避免在栈上分配大的数据结构，以免栈溢出，导致系统崩溃。kstack 记录了分配给该进程/线程的内核栈的位置。主要作用有以下几点。首先，当内核准备从一个进程切换到另一个进程的时候，需要根据 kstack 的值正确地设置好 tss(可以回顾一下在实验 1 中讲述的 tss 在中断处理过程中的作用)，以便在进程切换以后再发生中断时能够使用正确的栈。其次，内核栈位于内核地址空间，并且是不共享的(每个线程都拥有自己的内核栈)，因此不受 mm 的管理，当进程退出的时候，内核能够根据 kstack 的值快速定位栈的位置并进行回收。ucore 的这种内核栈的设计借鉴的是 Linux 的方法(但由于内存管理实现的差异，它实现的远不如 Linux 的灵活)，它使得每个线程的内核栈在不同的位置，这样从某种程度上方便调试，但同时也使得内核对栈溢出变得十分不敏感，因为一旦发生溢出，它极可能污染内核中其他的数据使得内核崩溃。如果能够通过页表，将所有进程的内核栈映射到固定的地址上，能够避免这种问题，但又会使得进程切换过程中对栈的修改变得相当烦琐。感兴趣的同学可以参考 Linux kernel 的代码对此进行尝试。

为了管理系统中所有的进程控制块，ucore 维护了如下全局变量(位于 kern/process/proc.c)。

① static struct proc * current：当前占用 CPU 且处于“运行”状态进程控制块指针。通常这个变量是只读的，只有在进程切换的时候才进行修改，并且整个切换和修改过程需要保证操作的原子性，目前至少需要屏蔽中断。可以参考 switch_to 的实现。

② static struct proc * initproc：本实验中，指向一个内核线程。本实验以后，此指针将指向第一个用户态进程。

③ static list_entry_t hash_list[HASH_LIST_SIZE]：所有进程控制块的散列表，proc_struct 中的成员变量 hash_link 将基于 pid 链接入这个散列表中。

④ list_entry_t proc_list：所有进程控制块的双向线性列表，proc_struct 中的成员变量 list_link 将链接入这个链表中。

5.3.3 创建并执行内核线程

建立进程控制块(proc.c 中的 alloc_proc 函数)后,现在就可以通过进程控制块来创建具体的进程了。首先,考虑最简单的内核线程,它通常只是内核中的一小段代码或者函数,没有用户空间。由于在操作系统启动后,已经对整个核心内存空间进行了管理,通过设置页表建立了核心虚拟空间(即 boot_cr3 指向的二级页表描述的空间)。所以内核中的所有线程都不需要再建立各自的页表,只需共享这个核心虚拟空间就可以访问整个物理内存。

1. 创建第 0 个内核线程 idleproc

init.c::kern_init 函数调用了 proc.c::proc_init 函数。proc_init 函数启动了创建内核线程的步骤。首先当前的执行上下文(从 kern_init 启动至今)就可以看成 ucore 内核(也可看做内核进程)中的一个内核线程的上下文。为此,ucore 通过给当前执行的上下文分配一个进程控制块以及对它进行相应初始化,将其打造成第 0 个内核线程——idleproc。具体步骤如下。

首先调用 alloc_proc 函数来通过 kmalloc 函数获得 proc_struct 结构的一块内存——proc,这就是第 0 个进程控制块了,并把 proc 进行初步初始化(即把 proc_struct 中的各个成员变量清零)。但有些成员变量设置了特殊的值:

```
练习 1            //设置进程为"初始"态
练习 1            //进程的 pid还没设置好
练习 1            //进程在内核中使用的内核页表的起始地址
```

上述三条语句中,第一条设置了进程的状态为"初始"态,这表示进程已经 "出生"了,正在获取资源茁壮成长中;第二条语句设置了进程的 pid 为-1,这表示进程的"身份证号"还没有办好;第三条语句表明由于该内核线程在内核中运行,故采用为 ucore 内核已经建立的页表,即设置为在 ucore 内核页表的起始地址 boot_cr3。后续实验中可进一步看出所有进程的内核虚地址空间(也包括物理地址空间)是相同的。既然内核线程共用一个映射内核空间的页表,这表示所有这些内核空间对所有内核线程都是"可见"的,所以更精确地说,这些内核线程都应该是从属于同一个唯一的内核进程——ucore 内核。

接下来,proc_init 函数对 idleproc 内核线程进行进一步初始化:

```
idleproc->pid=0;
idleproc->state=PROC_RUNNABLE;
idleproc->kstack=(uintptr_t)bootstack;
idleproc->need_resched=1;
set_proc_name(idleproc, "idle");
```

需要注意前 4 条语句。第一条语句给了 idleproc 合法的身份证号——0,这名正言顺地表明了 idleproc 是第 0 个内核线程。通常可以通过 pid 的赋值来表示线程的创建和身份确定。0 是第一个的表示方法是计算机领域所特有的,比如 C 语言定义的第一个数组元素的小标也是 0。第二条语句改变了 idleproc 的状态,使得它从"出生"转到了"准备工作",就差 ucore 调度它执行了。第三条语句设置了 idleproc 所使用的内核栈的起始地址。需要注意以后的其他线程的内核栈都需要通过分配获得,因为 ucore 启动时设置的内核栈直接分配

给 idleproc 使用了。第四条很重要，因为 ucore 希望当前 CPU 应该做更有用的工作，而不是运行 idleproc 这个“无所事事”的内核线程，所以把 idleproc－＞need_resched 设置为 1，结合 idleproc 的执行主体——cpu_idle 函数的实现，可以清楚地看出如果当前 idleproc 在执行，则只要此标志为 1，马上就调用 schedule 函数要求调度器切换其他进程执行。

2. 创建第 1 个内核线程 initproc

第 0 个内核线程的主要工作是完成内核中各个子系统的初始化，然后就通过执行 cpu_idle 函数开始过退休生活了。所以 ucore 接下来还需创建其他进程来完成各种工作，但 idleproc 内核子线程自己不想做，于是就通过调用 kernel_thread 函数创建了一个内核线程 init_main。在实验 4 中，这个子内核线程的工作就是输出一些字符串，然后就返回了（参见 init_main 函数）。但在后续的实验中，init_main 的工作就是创建特定的其他内核线程或用户进程（实验 5 涉及）。下面我们来分析一下创建内核线程的函数 kernel_thread：

```
kernel_thread(int (* fn)(void* ), void* arg, uint32_t clone_flags) {
    struct trapframe tf;
    memset(&tf, 0, sizeof(struct trapframe));
    tf.tf_cs=KERNEL_CS;
    tf.tf_ds=tf_struct.tf_es=tf_struct.tf_ss=KERNEL_DS;
    tf.tf_regs.reg_ebx=(uint32_t)fn;
    tf.tf_regs.reg_edx=(uint32_t)arg;
    tf.tf_eip=(uint32_t)kernel_thread_entry;
    return do_fork(clone_flags | CLONE_VM, 0, &tf);
}
```

注意：kernel_thread 函数采用了局部变量 tf 来放置保存内核线程的临时中断帧，并把中断帧的指针传递给 do_fork 函数，而 do_fork 函数会调用 copy_thread 函数来在新创建的进程内核栈上专门给进程的中断帧分配一块空间。

给中断帧分配完空间后，就需要构造新进程的中断帧，具体过程是：首先给 tf 进行清零初始化，并设置中断帧的代码段（tf. tf_cs）和数据段（tf. tf_ds/tf_es/tf_ss）为内核空间的段（KERNEL_CS/ KERNEL_DS），这实际上也说明了 initproc 内核线程在内核空间中执行。而 initproc 内核线程从哪里开始执行呢？tf. tf_eip 指出的是 kernel_thread_entry（位于 kern/process/entry. S 中），kernel_thread_entry 是 entry. S 中实现的汇编函数，它做的事情很简单：

```
kernel_thread_entry:            # void kernel_thread(void)
    pushl %edx                  # push arg
    call * %ebx                 # call fn
    pushl %eax                  # save the return value of fn(arg)
    call do_exit                # call do_exit to terminate current thread
```

从以上代码可以看出，kernel_thread_entry 函数主要为内核线程的主体 fn 函数做了一个准备开始和结束运行的“壳”，并把函数 fn 的参数 arg（保存在 edx 寄存器中）压栈，然后调用 fn 函数，把函数返回值 eax 寄存器内容压栈，调用 do_exit 函数退出线程执行。

do_fork 是创建线程的主要函数。kernel_thread 函数通过调用 do_fork 函数最终完成内核线程的创建工作。下面我们来分析一下 do_fork 函数的实现（练习 2）。do_fork 函数主

要做了以下 6 件事情。

（1）分配并初始化进程控制块（alloc_proc 函数）。

（2）分配并初始化内核栈（setup_stack 函数）。

（3）根据 clone_flag 标志复制或共享进程内存管理结构（copy_mm 函数）。

（4）设置进程在内核（将来也包括用户态）正常运行和调度所需的中断帧，以及执行上下文（copy_thread 函数）。

（5）把设置好的进程控制块放入 hash_list 和 proc_list 两个全局进程链表中。

（6）自此，进程已经准备好执行了，把进程状态设置为“就绪”态。

（7）设置返回码为子进程的 id 号。

这里需要注意的是，如果上述前 3 步执行没有成功，则需要做对应的出错处理，把相关已经占有的内存释放掉。copy_mm 函数目前只是把 current－＞mm 设置为 NULL，这是由于目前在实验 4 中只能创建内核线程，proc－＞mm 描述的是进程用户态空间的情况，所以目前 mm 还用不上。copy_thread 函数做的事情比较多，代码如下：

```
static void
copy_thread(struct proc_struct * proc, uintptr_t esp, struct trapframe * tf) {
    //在内核堆栈的顶部设置中断帧大小的一块栈空间
    proc->tf= (struct trapframe * ) (proc->kstack+ KSTACKSIZE) - 1;
    * (proc->tf)= * tf;             //复制在 kernel_thread 函数建立的临时中断帧的初始值
    proc->tf->tf_regs.reg_eax= 0;        //设置子进程/线程执行完 do_fork 后的返回值
    proc->tf->tf_esp= esp;               //设置中断帧中的栈指针 esp
    proc->tf->tf_eflags |= FL_IF;        //使能中断
    proc->context.eip= (uintptr_t)forkret;
    proc->context.esp= (uintptr_t) (proc->tf);
}
```

此函数首先在内核堆栈的顶部设置中断帧大小的一块栈空间，并在此空间中复制在 kernel_thread 函数建立的临时中断帧的初始值，并进一步设置中断帧中的栈指针 esp 和标志寄存器 eflags，特别是 eflags 设置了 FL_IF 标志，这表示此内核线程在执行过程中，能响应中断，打断当前的执行。执行到这步后，此进程的中断帧就建好了，对于 initproc 而言，它的中断帧如下：

```
//所在地址位置
initproc->tf= (proc->kstack+ KSTACKSIZE) - sizeof (struct trapframe);
//具体内容
initproc->tf.tf_cs= KERNEL_CS;
initproc->tf.tf_ds= initproc->tf.tf_es= initproc->tf.tf_ss= KERNEL_DS;
initproc->tf.tf_regs.reg_ebx= (uint32_t)init_main;
initproc->tf.tf_regs.reg_edx= (uint32_t) ADDRESS of "Hello?world!!";
initproc->tf.tf_eip= (uint32_t)kernel_thread_entry;
initproc->tf.tf_regs.reg_eax= 0;
initproc->tf.tf_esp= esp;
initproc->tf.tf_eflags |= FL_IF;
```

设置好中断帧后，最后就是设置 initproc 的进程上下文（process context，也称执行现场）了。只有设置好执行现场后，一旦 ucore 调度器选择了 initproc 执行，就需要根据 initproc－＞context 中保存的执行现场来恢复 initproc 的执行。这里设置了 initproc 的执行现场中主要的两个信息：上次停止执行时的下一条指令地址 context. eip 和上次停止执行时的堆栈地址 context. esp。其实 initproc 还没有执行过，所以这其实就是 initproc 实际执行的第一条指令地址和堆栈指针。可以看出，由于 initproc 的中断帧占用了实际给 initproc 分配的栈空间的顶部，所以 initproc 就只能把栈顶指针 context. esp 设置在 initproc 的中断帧的起始位置。根据 context. eip 的赋值，可以知道 initproc 实际开始执行的地方在 forkret 函数（主要完成 do_fork 函数返回的处理工作）处。至此，initproc 内核线程已经做好准备执行了。

3. 调度并执行内核线程 initproc

在 ucore 执行完 proc_init 函数后，就创建好了两个内核线程：idleproc 和 initproc，这时 ucore 当前的执行现场就是 idleproc，等到执行到 init 函数的最后一个函数 cpu_idle 之前，ucore 的所有初始化工作就结束了，idleproc 将通过执行 cpu_idle 函数让出 CPU，给其他内核线程执行，具体过程如下：

```
void
cpu_idle(void) {
    while(1){
        if(current->need_resched){
            schedule();
⋮
```

首先，判断当前内核线程 idleproc 的 need_resched 是否不为 0，回顾前面“创建第一个内核线程 idleproc”中的描述，proc_init 函数在初始化 idleproc 中，就把 idleproc－＞need_resched 置为 1 了，所以会马上调用 schedule 函数找其他处于“就绪”态的进程执行。

ucore 在实验 4 中只实现了一个最简单的 FIFO 调度器，其核心就是 schedule 函数。它的执行逻辑如下。

（1）设置当前内核线程 current－＞need_resched 为 0。

（2）在 proc_list 队列中查找下一个处于“就绪”态的线程或进程 next。

（3）找到这样的进程后，就调用 proc_run 函数，保存当前进程 current 的执行现场（进程上下文），恢复新进程的执行现场，完成进程切换。

至此，新的进程 next 就开始执行了。由于在 proc10 中只有两个内核线程，且 idleproc 要让出 CPU 给 initproc 执行，我们可以看到 schedule 函数通过查找 proc_list 进程队列，只能找到一个处于“就绪”态的 initproc 内核线程，并通过 proc_run 和进一步的 switch_to 函数完成两个执行现场的切换，具体流程如下。

（1）让 current 指向 next 内核线程 initproc。

（2）设置任务状态段 ts 中特权态 0 下的栈顶指针 esp0 为 next 内核线程 initproc 的内核栈的栈顶，即 next－＞kstack＋KSTACKSIZE 。

（3）设置 CR3 寄存器的值为 next 内核线程 initproc 的页目录表起始地址 next－＞cr3，这实际上是完成进程间的页表切换。

(4) 由 switch_to 函数完成具体的两个线程的执行现场切换，即切换各个寄存器，当 switch_to 函数执行完 ret 指令后，就切换到 initproc 执行了。

注意：在第(2)步设置任务状态段 ts 中特权态 0 下的栈顶指针 esp0 的目的是建好内核线程或将来用户线程在执行特权态切换(从特权态 0 转到特权态 3，或从特权态 3 转到特权态 3)时能够正确定位处于特权态 0 时进程的内核栈的栈顶，而这个栈顶其实放了一个 trapframe 结构的内存空间。如果是在特权态 3 发生了中断/异常/系统调用，则 CPU 会从特权态 3 转换特权态 0，且 CPU 从此栈顶(当前被打断进程的内核栈顶)开始压栈来保存被中断/异常/系统调用打断的用户态执行现场；如果是在特权态 0 发生了中断/异常/系统调用，则 CPU 会从当前内核栈指针 esp 所指的位置开始压栈保存被中断/异常/系统调用打断的内核态执行现场。反之，当执行完对中断/异常/系统调用打断的处理后，最后会执行一个 iret 指令。在执行此指令之前，CPU 的当前栈指针 esp 一定指向上次产生中断/异常/系统调用时 CPU 保存的被打断的指令地址 CS 和 EIP，iret 指令会根据 ESP 所指的保存的址 CS 和 EIP 恢复到上次被打断的地方继续执行。

在页表设置方面，由于 idleproc 和 initproc 都是共用一个内核页表 boot_cr3，所以此时第三步其实没用，但考虑到以后的进程有各自的页表，其起始地址各不相同，只有完成页表切换，才能确保新的进程能够正常执行。

第(4)步 proc_run 函数调用 switch_to 函数，参数是前一个进程和后一个进程的执行现场：process context。在 5.3.2 节中，描述了 context 结构包含的要保存和恢复的寄存器。下面再看看 switch.S 中的 switch_to 函数的执行流程：

```
.globl switch_to
switch_to:                      # switch_to(from, to)

    # save from's registers
    movl 4(%esp), %eax          # eax points to from
    popl 0(%eax)        # esp--> return address,   so save return address in FROM's context
    movl %esp, 4(%eax)
    ⋮
    movl %ebp, 28(%eax)         # restore to's registers
    movl 4(%esp), %eax          # not 8(%esp): popped return address already
                                # eax now points to to
    movl 28(%eax), %ebp
    ⋮
    movl 4(%eax), %esp
    pushl 0(%eax)               # push TO's context's eip, so return addr=TO's eip
    ret                         # after ret, eip=TO's eip
```

首先，保存前一个进程的执行现场，前两条汇编指令(如下所示)保存了进程在返回 switch_to 函数后的指令地址到 context.eip 中：

```
movl 4(%esp), %eax   # eax points to from
popl 0(%eax)         # esp--> return address,   so save return address in FROM's context
```

在接下来的 7 条汇编指令完成了保存前一个进程的其他 7 个寄存器到 context 中的相

应成员变量中。至此前一个进程的执行现场保存完毕。再往后是恢复向一个进程的执行现场，这其实就是上述保存过程的逆执行过程，即从 context 的高地址的成员变量 ebp 开始，逐一把相关成员变量的值赋值给对应的寄存器，倒数第二条汇编指令“pushl 0(%eax)”其实把 context 中保存的下一个进程要执行的指令地址 context. eip 放到了堆栈顶，这样接下来执行最后一条指令 ret 时，会把栈顶的内容赋值给 EIP 寄存器，这样就切换到下一个进程执行了，即当前进程已经是下一个进程。

ucore 会执行进程切换，让 initproc 执行。在对 initproc 进行初始化时，设置 initproc－>context. eip ＝ (uintptr_t)forkret，这样，当执行 switch_to 函数并返回后，initproc 将执行其实际上的执行入口地址 forkret。而 forkret 会调用位于 kern/trap/trapentry. S 中的 forkrets 函数执行，具体代码如下：

```
.globl __trapret
__trapret:
    # restore registers from stack
    popal
    # restore %ds and %es
    popl %es
    popl%ds
    # get rid of the trap number and error code
    addl $0x8,%esp
    iret
.globl forkrets
forkrets:
    # set stack to this new process's trapframe
    movl 4(%esp),%esp              //把 esp 指向当前进程的中断帧
    jmp__trapret
```

可以看出，forkrets 函数首先把 esp 指向当前进程的中断帧，从_trapret 开始执行到 iret 前，esp 指向了 current－>tf. tf_eip，而如果此时执行的是 initproc，则 current－>tf. tf_eip＝kernel_thread_entry，initproc－>tf. tf_cs ＝ KERNEL_CS，所以当执行完 iret 后，就开始在内核中执行 kernel_thread_entry 函数了，而 initproc－>tf. tf_regs. reg_ebx ＝ init_main，所以在 kernl_thread_entry 中执行“call %ebx”后，就开始执行 initproc 的主体了。initprocde 的主体函数就是输出一段字符串，然后就返回到 kernel_tread_entry 函数，并进一步调用 do_exit 执行退出操作了。本来 do_exit 应该完成一些资源回收工作等，但这些不是实验 4 涉及的，而是由后续的实验来完成。至此，实验 4 中的主要工作描述完毕。

5.4　实验报告要求

从网站上下载 lab4. zip 后，解压得到本文档和代码目录 lab4，完成实验中的各个练习。完成代码编写并检查无误后，在对应目录下执行 make handin 任务，即会自动生成 lab4-handin. tar. gz。最后请一定提前或按时提交到网络学堂上。

注意有 lab4 的注释，代码中所有需要完成的地方(Challenge 除外)都有 lab4 和“Your

Code"的注释，请在提交时特别注意保持注释，并将"Your Code"替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

辅助材料 A　实验 4 的参考输出

```
make qemu
(THU.CST)os is loading...

Special kernel symbols:
entry    0xc010002c (phys)
etext    0xc010d0f7 (phys)
edata    0xc012dad0 (phys)
end      0xc0130e78 (phys)
Kernel executable memory footprint: 196KB
memory management: default_pmm_manager
e820map:
memory: 0009f400, [00000000, 0009f3ff], type=1.
memory: 00000c00, [0009f400, 0009ffff], type=2.
memory: 00010000, [000f0000, 000fffff], type=2.
memory: 07efd000, [00100000, 07ffcfff], type=1.
memory: 00003000, [07ffd000, 07ffffff], type=2.
memory: 00040000, [fffc0000, ffffffff], type=2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
--------------------- BEGIN ---------------------
PDE(0e0) c0000000-f8000000 38000000 urw
|-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
|-- PTE(000e0) faf00000-fafe0000 000e0000 urw
|-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ---------------------
check_slab() succeeded!
kmalloc_init() succeeded!
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:      262144(sectors), 'QEMU HARDDISK'.
SWAP: manager = fifo swap manager
BEGIN check_swap: count 1, total 31944
mm->sm_priv c0130e64 in fifo_init_mm
setup Page Table for vaddr 0X1000, so alloc a page
```

```
setup Page Table vaddr 0~ 4MB OVER!
setup init env for check_swap begin!
page fault at 0x00001000: K/W [no page found].
page fault at 0x00002000: K/W [no page found].
page fault at 0x00003000: K/W [no page found].
page fault at 0x00004000: K/W [no page found].
set up init env for check_swap over!
write Virt Page c in fifo_check_swap
write Virt Page a in fifo_check_swap
write Virt Page d in fifo_check_swap
write Virt Page b in fifo_check_swap
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
write Virt Page b in fifo_check_swap
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
write Virt Page b in fifo_check_swap
page fault at 0x00002000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
write Virt Page c in fifo_check_swap
page fault at 0x00003000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in fifo_check_swap
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
check_swap() succeeded!
++ setup timer interrupts
this initproc, pid= 1, name= "init"
To U: "Hello world!!".
To U: "en.., Bye, Bye. :)"
kernel panic at kern/process/proc.c:316:
    process exit!!.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

辅助材料 B “原理”进程的属性与特征解析

操作系统负责进程管理，即从程序加载到运行结束的全过程，这个程序运行过程将经历从“出生”到“死亡”的完整“生命”历程。所谓“进程”就是指这个程序运行的整个执行过程。

为了记录、描述和管理程序执行的动态变化过程，需要有一个数据结构，这就是进程控制块。进程与进程控制块是一一对应的。为此，ucore 需要建立合适的进程控制块数据结构，并基于进程控制块来完成对进程的管理。

为了让多个程序能够使用 CPU 执行任务，需要设计用于进程管理的内核数据结构“进程控制块”。但到底如何设计进程控制块，如何管理进程？如果对进程的属性和特征了解不够，则无法有效地设计进程控制块和实现进程管理。

再一次回到进程的定义：一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。这里有四个关键词：程序、数据集合、执行和动态执行过程。从 CPU 的角度来看，程序就是一段特定的指令机器码序列而已。CPU 会一条一条地取出在内存中程序的指令并按照指令的含义执行各种功能；数据集合就是使用的内存；执行就是让 CPU 工作。这个数据集合和执行其实体现了进程对资源的占用。动态执行过程体现了程序执行的不同“生命”阶段：诞生、工作、休息/等待、死亡。如果这一段指令执行完毕，也就意味着进程结束了。从开始执行到执行结束是一个进程的全过程。那么操作系统需要管理进程的什么？如果计算机系统中只有一个进程，那么操作系统的工作就简单了。进程管理就是管理进程执行的指令，进程占用的资源，进程执行的状态。这可归结为对一个进程内的管理工作。但实际上在计算机系统的内存中，可以放很多程序，这也就意味着操作系统需要管理多个进程，那么，为了协调各进程对系统资源的使用，进程管理还需要做一些与进程协调有关的其他管理工作，包括进程调度、进程间的数据共享、进程间执行的同步互斥关系（后续相关实验涉及）等。下面逐一进行解析。

1. 资源管理

在计算机系统中，进程会占用内存和 CPU，这都是有限的资源，如果不进行合理的管理，资源会耗尽或无法高效公平地使用，从而会导致计算机系统中的多个进程执行效率很低，甚至由于资源不够而无法正常执行。

对于用户进程而言，操作系统是它的“上帝”，操作系统给了用户进程可以运行所需的资源，最基本的资源就是内存和 CPU。在实验 2 和实验 3 中涉及的内存管理方法和机制可直接应用到进程的内存资源管理中。在有多个进程存在的情况下，对于 CPU 这种资源，则需要通过进程调度来合理选择一个进程，并进一步通过进程分派和进程切换让不同的进程分时复用 CPU，执行各自的工作。对于无法剥夺的共享资源，如果资源管理不当，多个进程会出现死锁或饥饿现象。

2. 进程状态管理

用户进程有不同的状态（可理解为“生命”的不同阶段），当操作系统把程序的放到内存中后，这个进程就“诞生”了，不过还没有开始执行，但已经消耗了内存资源，处于“创建”状态；当进程准备好各种资源，就等能够使用 CPU 时，进程处于“就绪”状态；当进程终于占用 CPU，程序的指令被 CPU 一条一条执行的时候，这个进程就进入了“运行”状态，这时除了继续占用内存资源外，还占用了 CPU 资源；当进程由于等待某个资源而无法继续执行时，进程可放弃 CPU 使用，即释放 CPU 资源，进入“等待”状态；当程序指令执行完毕，由操作系统回收进程所占用的资源时，进程进入了“死亡”状态。

这些进程状态的转换时机需要操作系统管理起来，而且进程的创建和清除等服务必须由操作系统提供，而且在“运行”与“就绪”/“等待”状态之间的转换，涉及保存和恢复进程的

“执行现场”，也就是进程上下文，这是确保进程即使“断断续续”地执行，也能正确完成工作的必要保证。

3. 进程与线程

一个进程拥有一个存放程序和数据的虚拟地址空间和其他资源。一个进程基于程序的指令流执行，其执行过程可能与其他进程的执行过程交替进行。因此，一个具有执行状态（运行态、就绪态等）的进程是一个被操作系统分配资源（比如分配内存）并调度（比如分时使用 CPU）的单位。在大多数操作系统中，这两个特点是进程的主要本质特征。但这两个特征相对独立，操作系统可以把这两个特征分别进行管理。

这样可以把拥有资源所有权的单位通常仍称为进程，对资源的管理成为进程管理；把指令执行流的单位称为线程，对线程的管理就是线程调度和线程分派。对属于同一进程的所有线程而言，这些线程共享进程的虚拟地址空间和其他资源，但每个线程都有一个独立的栈，还有独立的线程运行上下文，用于包含表示线程执行现场的寄存器值等信息。

在多线程环境中，进程被定义成资源分配与保护的单位，与进程相关联的信息主要有存放进程映像的虚拟地址空间等。在一个进程中，可能有一个或多个线程，每个线程有线程执行状态（运行、就绪、等待等），保存上次运行时的线程上下文、线程的执行栈等。考虑到 CPU 有不同的特权模式，参照进程的分类，线程又可进一步细化为用户线程和内核线程。

到目前为止，我们就可以明确用户进程、内核进程（可把 ucore 看成一个内核进程）、用户线程、内核线程的区别了。从本质上看，线程就是一个特殊的不用拥有资源的轻量级进程，在 ucore 的调度和执行管理中，并没有区分线程和进程，且由于 ucore 内核中的所有内核线程共享一个内核地址空间和其他资源，所以这些内核线程从属于同一个唯一的内核进程，即 ucore 内核本身。理解了进程或线程的上述属性和特征，就可以进行进程/线程管理的设计与实现了。但是为了叙述上的简便，以下用户态的进程/线程统称为用户进程。

第 6 章　实验 5：用户进程管理

6.1 实验目的

（1）了解第一个用户进程创建的过程。

（2）了解系统调用框架的实现机制。

（3）了解 ucore 如何实现系统调用 sys_fork/sys_exec/sys_exit/sys_wait 来进行进程管理。

6.2 实验内容

实验 4 完成了内核线程，但到目前为止，所有的运行都在内核态执行。实验 5 将创建用户进程，让用户进程在用户态执行，且在需要 ucore 支持时，可通过系统调用来让 ucore 提供服务。为此需要构造出第一个用户进程，并通过系统调用 sys_fork/sys_exec/sys_exit/sys_wait 来支持运行不同的应用程序，完成对用户进程的执行过程的基本管理。相关原理介绍可看本章附录 B。

6.2.1 练习

练习 0：填写已有实验。

本实验依赖实验 1～实验 4。请把已做的实验 1～实验 4 的代码填入本实验中代码中有 lab1、lab2、lab3、lab4 的注释相应部分。

注意：*为了能够正确执行 lab5 的测试应用程序，可能需对已完成的实验 1～实验 4 的代码进行进一步改进。*

练习 1：加载应用程序并执行（需要编码）。

do_execv 函数调用 load_icode（位于 kern/process/proc.c 中）来加载并解析一个处于内存中的 ELF 执行文件格式的应用程序，建立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好 proc_struct 结构中的成员变量 trapframe 中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。需设置正确的 trapframe 内容。

练习 2：父进程复制自己的内存空间给子进程（需要编码）。

创建子进程的函数 do_fork 在执行中将复制当前进程（即父进程）的用户内存地址空间中的合法内容到新进程中（子进程），完成内存资源的复制。具体是通过 copy_range 函数（位于 kern/mm/pmm.c 中）实现的，请补充 copy_range 的实现，确保能够正确执行。

练习 3：阅读分析源代码，理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的

实现(不需要编码)。

执行：make grade。如果所显示的应用程序检测都输出 ok，则基本正确(使用的是 qemu-1.0.1)。

扩展练习 Challenge：实现 Copy on Write 机制。

这个扩展练习涉及本实验和实验“虚拟内存管理”。Copy on write(COW)的基本概念是指如果有多个使用者对同一个资源 A(比如内存块)进行读操作，则每个使用者只需获得一个指向同一个资源 A 的指针，就可以读该资源了。若某使用者需要对这个资源 A 进行写操作，系统会对该资源进行复制操作，从而使得该“写操作”使用者获得一个该资源 A 的“私有”副本——资源 B，可对资源 B 进行写操作。该“写操作”使用者对资源 B 的改变对于其他的使用者而言是不可见的，因为其他使用者看到的还是资源 A。

在 ucore 操作系统中，当一个用户父进程创建自己的子进程时，父进程会把其申请的用户空间设置为只读，子进程可共享父进程占用的用户内存空间中的页面(这就是一个共享的资源)。当其中任何一个进程修改此用户内存空间中的某页面时，ucore 会通过 page fault 异常获知该操作，并完成复制内存页面，使得两个进程都有各自的内存页面。这样一个进程所做的修改不会被另外一个进程可见。请在 ucore 中实现这样的 COW 机制。

6.2.2 项目组成

目录结构图如图 6-1 所示。

相对于实验 4，实验 5 主要增加的文件有 syscall.c、syscall.h、unistd.h、user.ld、hello.c 和 initcode.s 等，主要修改的文件有 kdebug.c、memlayout.h、pmm.c、pmm.h、vmm.c、vmm.h、proc.c、proc.h、sched.c、sync.h、elf.h、error.h、printfmt.c 等。主要改动如下。

1. kern/debug/

kdebug.c：修改，解析用户进程的符号信息表示(可不用理会)。

2. kern/mm/(与本次实验有较大关系)

memlayout.h：修改，增加了用户虚存地址空间的图形表示和宏定义(需仔细理解)。

pmm.[ch]：修改，添加了用于进程退出(do_exit)的内存资源回收的 page_remove_pte、unmap_range、exit_range 函数和用于创建子进程(do_fork)中复制父进程内存空间的 copy_range 函数，修改了 pgdir_alloc_page 函数。

vmm.[ch]：修改，扩展了 mm_struct 数据结构，增加了一系列函数。

(1) mm_map/dup_mmap/exit_mmap：设定/取消/复制/删除用户进程的合法内存空间。

(2) copy_from_user/copy_to_user：用户内存空间内容与内核内存空间内容的相互复制的实现。

(3) user_mem_check：搜索 vma 链表，检查是否是一个合法的用户空间范围。

3. kern/process/(与本次实验有较大关系)

proc.[ch]：修改，扩展了 proc_struct 数据结构，增加或修改了一系列函数。

(1) setup_pgdir/put_pgdir：创建并设置/释放页目录表。

(2) copy_mm：复制用户进程的内存空间和设置相关内存管理(如页表等)信息。

(3) do_exit：释放进程自身所占内存空间和相关内存管理(如页表等)信息所占空间，

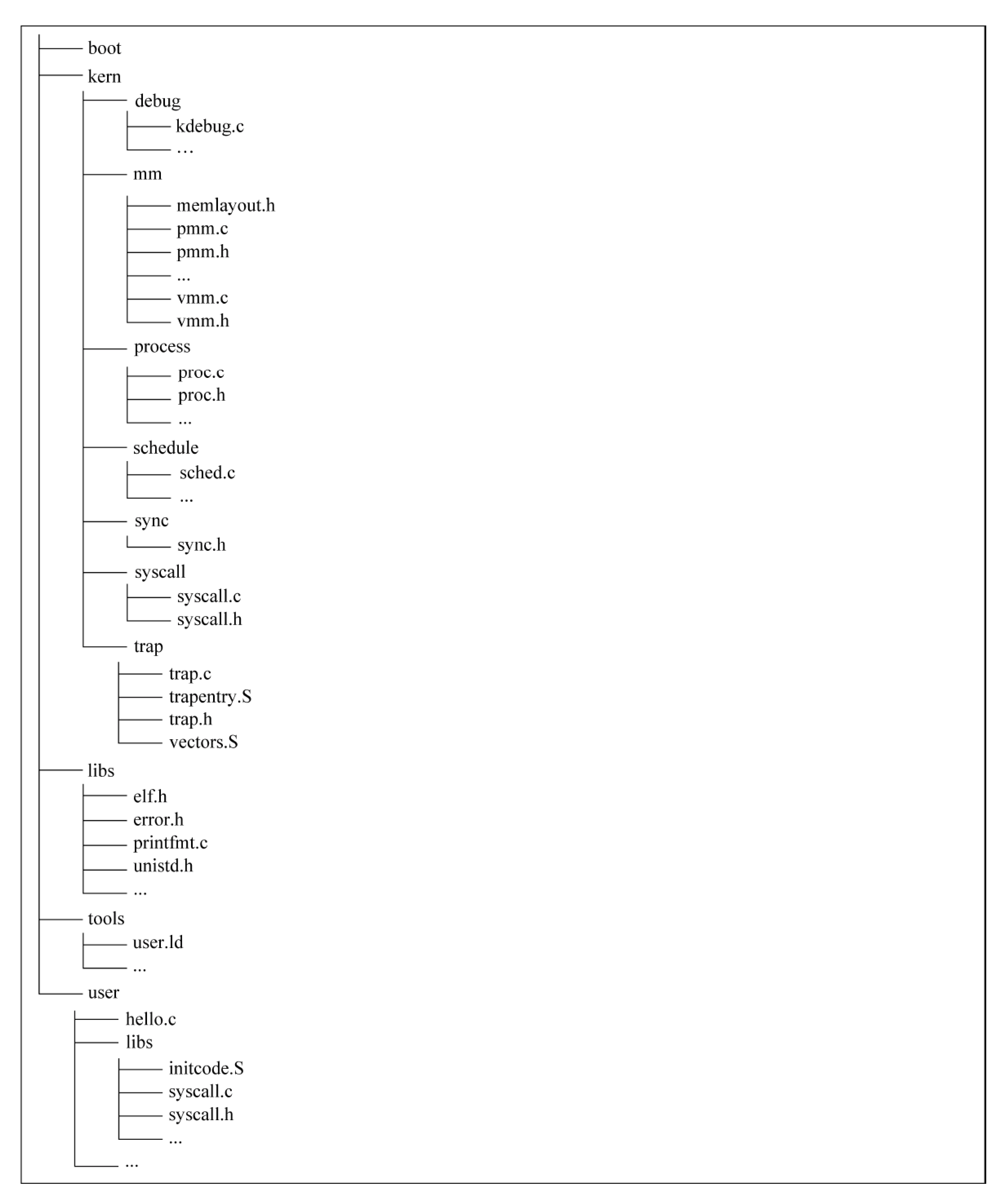


图 6-1　目录结构图

唤醒父进程，好让父进程收了自己，让调度器切换到其他进程。

（4）load_icode：被 do_execve 调用，完成加载放在内存中的执行程序到进程空间，这涉及对页表等的修改，分配用户栈。

（5）do_execve：先回收自身所占用户空间，然后调用 load_icode，用新的程序覆盖内存空间，形成一个执行新程序的新进程。

(6) do_yield：让调度器执行一次选择新进程的过程。

(7) do_wait：父进程等待子进程，并在得到子进程的退出消息后，彻底回收子进程所占的资源(比如子进程的内核栈和进程控制块)。

(8) do_kill：给一个进程设置 PF_EXITING 标志(kill 信息，即要它死掉)，这样在 trap 函数中，将根据此标志，让进程退出。

(9) KERNEL_EXECVE/__KERNEL_EXECVE/__KERNEL_EXECVE2：被 user_main 调用，执行一用户进程。

4. kern/trap/

trap.c：修改，在 idt_init 函数中，对 IDT 初始化时，设置好了用于系统调用的中断门(idt[T_SYSCALL])信息。这主要与 syscall 的实现相关。

5. user/ *

新增的用户程序和用户库。

6.3 用户进程管理

6.3.1 实验执行流程概述

到实验 4 为止，ucore 还一直在核心态“打转”，没有到用户态执行。提供各种操作系统功能的内核线程只能在 CPU 核心态运行是操作系统自身的要求，操作系统就要待在核心态，才能管理整个计算机系统。但应用程序员也需要编写各种应用软件，且要在计算机系统上运行。如果把这些应用软件都作为内核线程来执行，那系统的安全性就无法得到保证了。所以，ucore 要提供用户态进程的创建和执行机制，给应用程序执行提供一个用户态运行环境。接下来我们就简要分析本实验的执行过程，以及分析用户进程的整个生命周期来阐述用户进程管理的设计与实现。

显然，由于进程的执行空间扩展到了用户态空间，且出现了创建子进程执行应用程序等与 lab4 有较大不同的地方，所以具体实现的不同主要集中在进程管理和内存管理部分。首先，从 ucore 的初始化部分来看，会发现初始化的总控函数 kern_init 没有任何变化。但这并不意味着 lab4 与 lab5 的差别不大。其实 kern_init 调用的物理内存初始化，进程管理初始化等都有一定的变化。

在内存管理部分，与 lab4 最大的区别就是增加了用户态虚拟内存的管理。为了管理用户态的虚拟内存，需要对页表的内容进行扩展，能够把部分物理内存映射为用户态虚拟内存。如果某进程执行过程中，CPU 在用户态下执行(在 CS 段寄存器最低两位包含一个 2 位的优先级域，如果为 0，表示 CPU 运行在特权态；如果为 3，表示 CPU 运行在用户态)，则可以访问本进程页表描述的用户态虚拟内存，但由于权限不够，不能访问内核态虚拟内存。另外，不同的进程有各自的页表，所以即使不同进程的用户态虚拟地址相同，但由于页表把虚拟页映射到了不同的物理页帧，所以不同进程的虚拟内存空间是被隔离开的，相互之间无法直接访问。在用户态内存空间和内核态内核空间之间需要复制数据，让 CPU 处在内核态才能完成对用户空间的读或写，为此需要设计专门的复制函数(copy_from_user 和 copy_to_user)完成。但反之则会导致违反 CPU 的权限管理，导致内存访问异常。

在进程管理方面，主要涉及的是进程控制块中与内存管理相关的部分，包括建立进程的页表和维护进程可访问空间（可能还没有建立虚实映射关系）的信息；加载一个 ELF 格式的程序到进程控制块管理的内存中的方法；在进程复制（fork）过程中，把父进程的内存空间复制到子进程内存空间的技术。另外一部分与用户态进程生命周期管理相关，包括让进程放弃 CPU 而睡眠等待某事件；让父进程等待子进程结束；一个进程杀死另一个进程；给进程发送消息；建立进程的血缘关系链表。

当实现了上述内存管理和进程管理的需求后，接下来 ucore 的用户进程管理工作就比较简单了。首先，“硬”构造出第一个进程（lab4 中已有描述），它是后续所有进程的祖先；然后，在 proc_init 函数中，通过 alloc 把当前 ucore 的执行环境转变成 idle 内核线程的执行现场；然后调用 kernl_thread 来创建第二个内核线程 init_main，而 init_main 内核线程又创建了 user_main 内核线程。到此，内核线程创建完毕，应该开始用户进程的创建过程，这一步实际上是通过 user_main 函数调用 kernel_tread 创建子进程，通过 kernel_execve 调用来把某一具体程序的执行内容放入内存。具体的放置方式是根据 ld 在此文件上的地址分配为基本原则，把程序的不同部分放到某进程的用户空间中，从而通过此进程来完成程序描述的任务。一旦执行了这一程序对应的进程，就会从内核态切换到用户态继续执行。以此类推，CPU 在用户空间执行的用户进程，其地址空间不会被其他用户的进程影响，但由于系统调用（用户进程直接获得操作系统服务的唯一通道）、外设中断和异常中断的会随时产生，从而间接推动了用户进程实现用户态到内核态的切换工作。ucore 对 CPU 内核态与用户态的切换过程需要比较仔细地分析（这其实是实验 1 的扩展练习）。当进程执行结束后，需回收进程占用和没消耗完毕的设备整个过程，且为新的创建进程请求提供服务。在本实验中，当系统中存在多个进程或内核线程时，ucore 采用了一种 FIFO 的很简单的调度方法来管理每个进程占用 CPU 的时间和频度等。在 ucore 运行过程中，由于调度、时间中断、系统调用等原因，使得进程会进行切换、创建、睡眠、等待、发送消息等各种不同的操作，周而复始，生生不息。

6.3.2 创建用户进程

在实验 4 中，已经完成了对内核线程的创建，但与用户进程的创建过程相比，创建内核线程的过程还远远不够。这两个创建过程的差异本质上就是用户进程和内核线程的差异决定的。

1. 应用程序的组成和编译

我们首先来看一个应用程序，这里假定是 hello 应用程序，在 user/hello.c 中实现，代码如下：

```
#include <stdio.h>
#include<ulib.h>
int
main(void) {
    cprintf("Hello world!!.\n");
    cprintf("I am process %d.\n",getpid());
    cprintf("hello pass.\n");
return0;
}
```

hello 应用程序只是输出一些字符串，并通过系统调用 sys_getpid（在 getpid 函数中调

用)输出代表 hello 应用程序执行的用户进程的进程标识——pid。

首先,我们需要了解 ucore 操作系统如何能够找到 hello 应用程序。这需要分析 ucore 和 hello 是如何编译的。修改 Makefile,把第 6 行注释掉。然后在本实验源码目录下执行 make,可得到如下输出:

```
⋮
+ cc user/hello.c

gcc- Iuser/- fno- builtin- Wall- ggdb- m32- gstabs- nostdinc - fno- stack- protector- Ilibs/- Iuser/include/
- Iuser/libs/- c user/hello.c- o obj/user/hello.o

ld- m    elf_i386- nostdlib- T tools/user.ld- o obj/__user_hello.out   obj/user/libs/initcode.o obj/
user/libs/panic.o  obj/user/libs/stdio.o  obj/user/libs/syscall.o  obj/user/libs/ulib.o  obj/user/libs/
umain.o   obj/libs/hash.o obj/libs/printfmt.o obj/libs/rand.o obj/libs/string.o obj/user/hello.o
⋮
ld- m    elf_i386- nostdlib- T tools/kernel.ld- o bin/kernel   obj/kern/init/entry.o obj/kern/init/init.o
… - b binary… obj/__user_hello.out
⋮
```

从中可以看出,hello 应用程序不仅仅是 hello.c,还包含了支持 hello 应用程序的用户态库。

(1) user/libs/initcode.S:所有应用程序的起始用户态执行地址_start,调整了 EBP 和 ESP 后,调用 umain 函数。

(2) user/libs/umain.c:实现了 umain 函数,这是所有应用程序执行的第一个 C 函数,它将调用应用程序的 main 函数,并在 main 函数结束后调用 exit 函数,而 exit 函数最终将调用 sys_exit 系统调用,让操作系统回收进程资源。

(3) user/libs/ulib.[ch]:实现了最小的 C 函数库,除了一些与系统调用无关的函数,其他函数是对访问系统调用的包装。

(4) user/libs/syscall.[ch]:用户层发出系统调用的具体实现。

(5) user/libs/stdio.c:实现 cprintf 函数,通过系统调用 sys_putc 来完成字符输出。

(6) user/libs/panic.c:实现__panic/__warn 函数,通过系统调用 sys_exit 完成用户进程退出。

除了这些用户态库函数实现外,还有一些 libs/*.[ch]是操作系统内核和应用程序共用的函数实现。这些用户库函数其实在本质上与 UNIX 系统中的标准 libc 没有区别,只是实现得很简单,但 hello 应用程序的正确执行离不开这些库函数。

注意:libs/*.[ch]、user/libs/*.[ch]、user/*.[ch]的源码中没有任何特权指令。

在 make 的最后一步执行了一个 ld 命令,把 hello 应用程序的执行码 obj/__user_hello.out 连接在了 ucore kernel 的末尾。且 ld 命令会在 kernel 中把__user_hello.out 的位置和大小记录在全局变量_binary_obj___user_hello_out_start 和_binary_obj___user_hello_out_size 中,这样这个 hello 用户程序就能够和 ucore 内核一起被 bootloader 加载到内存中,并且通过这两个全局变量定位 hello 用户程序执行码的起始位置和大小。到了与文件系统相关的实验后,ucore 会提供一个简单的文件系统,那时所有的用户程序就都不再用这种方法进行加载,而可以用大家熟悉的文件方式进行加载。

2. 用户进程的虚拟地址空间

在 tools/user.ld 描述了用户程序的用户虚拟空间的执行入口虚拟地址：

```
SECTIONS {
    /* Load programs at this address: "." means the current address */
    . = 0x800020;
```

在 tools/kernel.ld 描述了操作系统的内核虚拟空间的起始入口虚拟地址：

```
SECTIONS {
    /* Load the kernel at this address: "." means the current address */
    . = 0xC0100000;
```

这样 ucore 把用户进程的虚拟地址空间分了两块，一块与内核线程一样，是所有用户进程都共享的内核虚拟地址空间，映射到同样的物理内存空间中，这样在物理内存中只需放置一份内核代码，使得用户进程从用户态进入核心态时，内核代码可以统一应对不同的内核程序；另外一块是用户虚拟地址空间，虽然虚拟地址范围一样，但映射到不同且没有交集的物理内存空间中。这样当 ucore 把用户进程的执行代码(即应用程序的执行代码)和数据(即应用程序的全局变量等)放到用户虚拟地址空间中时，确保了各个进程不会"非法"访问到其他进程的物理内存空间。

这样 ucore 给一个用户进程具体设定的虚拟内存空间(kern/mm/memlayout.h)如图 6-2 所示。

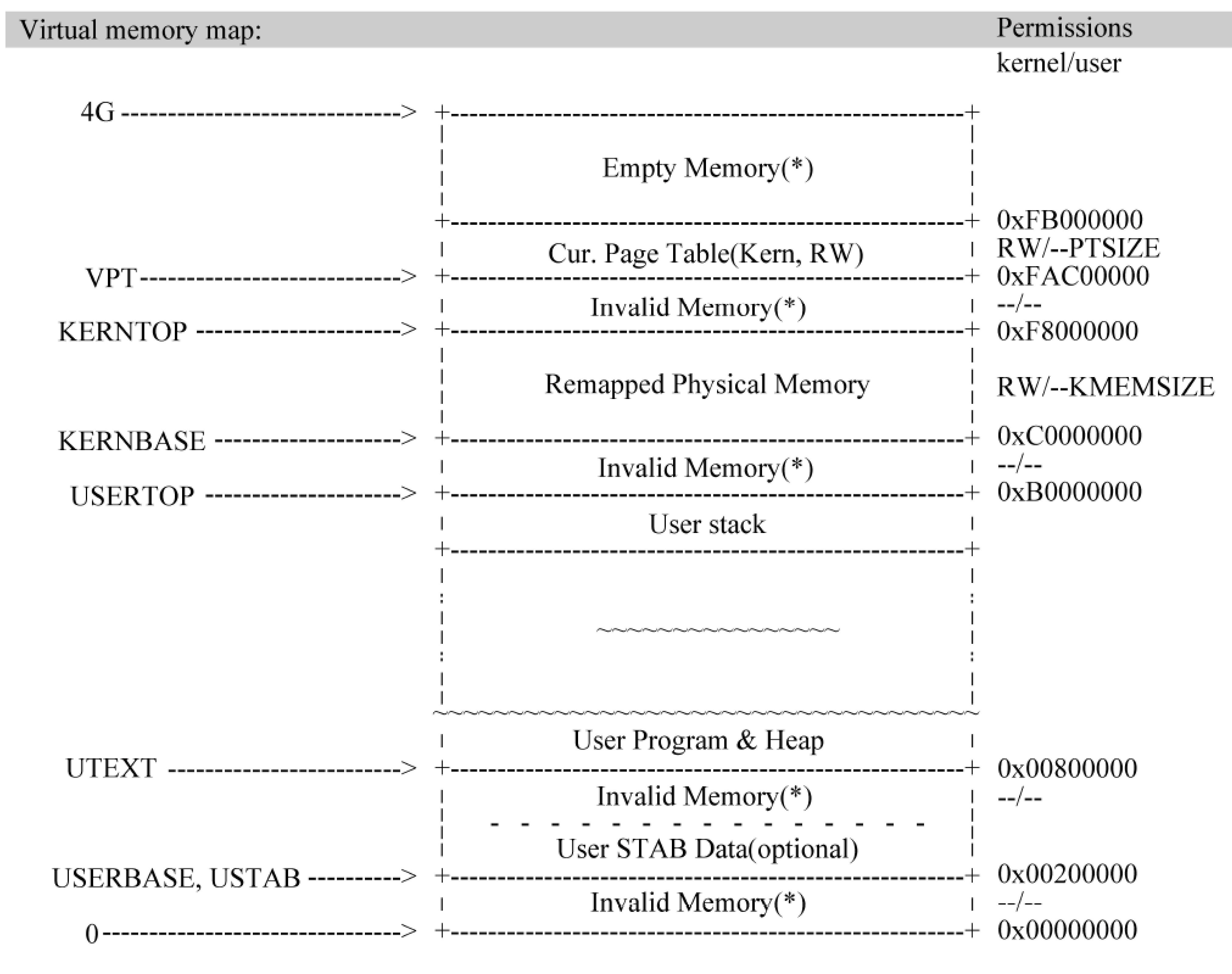


图 6-2 用户进程的虚拟内存空间

3. 创建并执行用户进程

在确定了用户进程的执行代码和数据，以及用户进程的虚拟空间布局后，我们可以来创建用户进程了。在本实验中第一个用户进程是由第二个内核线程 initproc 通过把 hello 应用程序执行码覆盖到 initproc 的用户虚拟内存空间来创建的，相关代码如下：

```
//kernel_execve - do SYS_exec syscall to exec a user program called by user_main kernel_thread
static int
kernel_execve(const char * name, unsigned char * binary, size_t size) {
    int ret, len = strlen(name);
    asm volatile (
        "int %1;"
        : "=a" (ret)
        : "i" (T_SYSCALL), "0" (SYS_exec), "d" (name), "c" (len), "b" (binary), "D" (size)
        : "memory");
    return ret;
}

#define __KERNEL_EXECVE(name, binary, size) ({
            cprintf("kernel_execve: pid = %d, name = \"%s\".\n",
                current->pid, name);
            kernel_execve(name, binary, (size_t)(size));
        })

#define KERNEL_EXECVE(x) ({
            extern unsigned char _binary_obj__user_##x##_out_start[],
                _binary_obj__user_##x##_out_size[];
            __KERNEL_EXECVE(#x, _binary_obj__user_##x##_out_start,
                        _binary_obj__user_##x##_out_size);
        })
⋮
//init_main - the second kernel thread used to create kswapd_main & user_main kernel threads
static int
init_main(void * arg) {
#ifdef TEST
    KERNEL_EXECVE2(TEST, TESTSTART, TESTSIZE);
#else
    KERNEL_EXECVE(hello);
#endif
    panic("kernel_execve failed.\n");
    return 0;
}
```

对于上述代码，我们需要从后向前按照函数/宏的实现一个一个来分析。initproc 的执行主体是 init_main 函数，这个函数在默认情况下是执行宏 KERNEL_EXECVE(hello)，而这个宏最终是调用 kernel_execve 函数来调用 SYS_exec 系统调用，ld 在链接 hello 应用程

序执行码时定义了两全局变量。

（1）_binary_obj___user_hello_out_start：hello 执行码的起始位置。

（2）_binary_obj___user_hello_out_size 中：hello 执行码的大小。

kernel_execve 把这两个变量作为 SYS_exec 系统调用的参数，让 ucore 来创建此用户进程。当 ucore 收到此系统调用后，将依次调用如下函数：

```
vector128(vectors.S)-->__alltraps(trapentry.S)-->trap(trap.c)-->trap_dispatch(trap.c)--
-->syscall(syscall.c)-->sys_exec(syscall.c)-->do_execve(proc.c)
```

最终通过 do_execve 函数来完成用户进程的创建工作。此函数的主要工作流程如下。

（1）为加载新的执行码做好用户态内存空间清空准备。如果 mm 不为 NULL，则设置页表为内核空间页表，且进一步判断 mm 的引用计数减 1 后是否为 0，如果为 0，则表明没有进程再需要此进程所占用的内存空间，为此将根据 mm 中的记录，释放进程所占用户空间内存和进程页表本身所占空间。最后把当前进程的 mm 内存管理指针为空。由于此处的 initproc 是内核线程，所以 mm 为 NULL，整个处理都不会做。

（2）加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。这里涉及读 ELF 格式的文件，申请内存空间，建立用户态虚存空间，加载应用程序执行码等。load_icode 函数完成整个复杂的工作。

load_icode 函数的主要工作就是给用户进程建立一个能够让用户进程正常运行的用户环境。此函数有 100 多行，完成了如下重要工作。

（1）调用 mm_create 函数来申请进程的内存管理数据结构 mm 所需内存空间，并对 mm 进行初始化。

（2）调用 setup_pgdir 来申请一个页目录表所需的一个页大小的内存空间，并把描述 ucore 内核虚空间映射的内核页表（boot_pgdir 所指）的内容复制到此新目录表中，最后让 mm－＞pgdir 指向此页目录表，这就是进程新的页目录表了，且能够正确映射内核虚空间。

（3）根据应用程序执行码的起始位置来解析此 ELF 格式的执行程序，并调用 mm_map 函数根据 ELF 格式的执行程序说明的各个段（代码段、数据段、BSS 段等）的起始位置和大小建立对应的 vma 结构，并把 vma 插入 mm 结构中，从而表明了用户进程的合法用户态虚拟地址空间。

（4）调用根据执行程序各个段的大小分配物理内存空间，并根据执行程序各个段的起始位置确定虚拟地址，并在页表中建立好物理地址和虚拟地址的映射关系，然后把执行程序各个段的内容复制到相应的内核虚拟地址中，至此应用程序执行码和数据已经根据编译时设定地址放置到虚拟内存中了。

（5）需要给用户进程设置用户栈，为此调用 mm_mmap 函数建立用户栈的 vma 结构，明确用户栈的位置在用户虚空间的顶端，大小为 256 个页，即 1MB，并分配一定数量的物理内存且建立好栈的虚地址＜－－－＞物理地址映射关系。

（6）至此，进程内的内存管理 vma 和 mm 数据结构已经建立完成，于是把 mm－＞pgdir 赋值到 cr3 寄存器中，即更新了用户进程的虚拟内存空间，此时的 initproc 已经被 hello 的代码和数据覆盖，成为了第一个用户进程，但此时这个用户进程的执行现场还没建好。

（7）先清空进程的中断帧，再重新设置进程的中断帧，使得在执行中断返回指令 iret 后，能够让 CPU 转到用户态特权级，并回到用户态内存空间，使用用户态的代码段、数据段和堆栈，且能够跳转到用户进程的第一条指令执行，并确保在用户态能够响应中断。

至此，用户进程的用户环境已经搭建完毕。此时 initproc 将按产生系统调用的函数调用路径原路返回，执行中断返回指令 iret（位于 trapentry.S 的最后一句）后，将切换到用户进程 hello 的第一条语句位置_start 处（位于 user/libs/initcode.S 的第三句）开始执行。

6.3.3 进程退出和等待进程

当进程执行完它的工作后，就需要执行退出操作，释放进程占用的资源。ucore 分两步来完成这项工作，首先由进程本身完成大部分资源的占用内存回收工作，然后由此进程的父进程完成剩余资源占用内存的回收工作。为何不让进程本身完成所有的资源回收工作呢？这是因为进程要执行回收操作，就表明此进程还存在，还在执行指令，这就需要内核栈的空间不能释放，且表示进程存在的进程控制块不能释放。所以需要父进程来帮忙释放子进程无法完成的这两个资源回收工作。

为此在用户态的函数库中提供了 exit 函数，此函数最终访问 sys_exit 系统调用接口让操作系统来帮助当前进程执行退出过程中的部分资源回收。下面来看看 ucore 是如何做进程退出工作的。

首先，exit 函数会把一个退出码 error_code 传递给 ucore，ucore 通过执行内核函数 do_exit 来完成对当前进程的退出处理，主要工作简单地说就是回收当前进程所占的大部分内存资源，并通知父进程完成最后的回收工作，具体流程如下。

（1）如果 current－＞mm ！＝ NULL，表示是用户进程，则开始回收此用户进程所占用的用户态虚拟内存空间。

① 执行 lcr3（boot_cr3），切换到内核态的页表上，这样当前用户进程目前只能在内核虚拟地址空间执行了，这是为了确保后续释放用户态内存和进程页表的工作能够正常执行。

② 如果当前进程控制块的成员变量 mm 的成员变量 mm_count 减 1 后为 0（表明这个 mm 没有再被其他进程共享，可以彻底释放进程所占的用户虚拟空间了），则开始回收用户进程所占的内存资源。

a. 调用 exit_mmap 函数释放 current－＞mm－＞vma 链表中每个 vma 描述的进程合法空间中实际分配的内存，然后把对应的页表项内容清空，最后还把页表所占用的空间释放并把对应的页目录表项清空。

b. 调用 put_pgdir 函数释放当前进程的页目录所占的内存。

c. 调用 mm_destroy 函数释放 mm 中的 vma 所占内存，最后释放 mm 所占内存。

③ 此时设置 current－＞mm 为 NULL，表示与当前进程相关的用户虚拟内存空间和对应的内存管理成员变量所占的内核虚拟内存空间已经回收完毕。

（2）这时，设置当前进程的执行状态 current－＞state＝PROC_ZOMBIE 和当前进程的退出码 current－＞exit_code＝error_code。此时当前进程已经不能被调度了，需要此进程的父进程来做最后的回收工作（即回收描述此进程的内核栈和进程控制块）。

（3）如果当前进程的父进程 current－＞parent 处于等待子进程状态：

```
current->parent->wait_state==WT_CHILD
```

则唤醒父进程(即执行 wakup_proc(current－＞parent)),让父进程帮助自己完成最后的资源回收。

(4) 如果当前进程还有子进程,则需要把这些子进程的父进程指针设置为内核线程 initproc,且各个子进程指针需要插入到 initproc 的子进程链表中。如果某个子进程的执行状态是 PROC_ZOMBIE,则需要唤醒 initproc 来完成对此子进程的最后回收工作。

(5) 执行 schedule()函数,选择新的进程执行。

那么父进程如何完成对子进程的最后回收工作呢？这要求父进程要执行 wait 用户函数或 wait_pid 用户函数,这两个函数的区别是,wait 函数等待任意子进程的结束通知,而 wait_pid 函数等待进程 id 号为 pid 的子进程结束通知。这两个函数最终访问 sys_wait 系统调用接口让 ucore 来完成对子进程的最后回收工作,即回收子进程的内核栈和进程控制块所占内存空间,具体流程如下。

(1) 如果 pid！＝0,表示只找一个进程 id 号为 pid 的退出状态的子进程,否则找任意一个处于退出状态的子进程。

(2) 如果此子进程的执行状态不为 PROC_ZOMBIE,表明此子进程还没有退出,则当前进程只好设置自己的执行状态为 PROC_SLEEPING,睡眠原因为 WT_CHILD(即等待子进程退出),调用 schedule()函数选择新的进程执行,自己睡眠等待,如果被唤醒,则重复跳回步骤(1)处执行。

(3) 如果此子进程的执行状态为 PROC_ZOMBIE,表明此子进程处于退出状态,需要当前进程(即子进程的父进程)完成对子进程的最终回收工作,即首先把子进程控制块从两个进程队列 proc_list 和 hash_list 中删除,并释放子进程的内核堆栈和进程控制块。自此,子进程才彻底地结束了它的执行过程,释放了它所占用的所有资源。

6.3.4 系统调用实现

系统调用的英文名字是 System Call。操作系统为什么需要实现系统调用呢？其实这是实现了用户进程后,自然引申出来需要实现的操作系统功能。用户进程只能在操作系统给它圈定好的“用户环境”中执行,但“用户环境”限制了用户进程能够执行的指令,即用户进程只能执行一般的指令,无法执行特权指令。如果用户进程想执行一些需要特权指令的任务,比如通过网卡发网络包等,只能让操作系统来代劳了。于是就需要一种机制来确保用户进程不能执行特权指令,但能够请操作系统“帮忙”完成需要特权指令的任务,这种机制就是系统调用。

采用系统调用机制为用户进程提供一个获得操作系统服务的统一接口层,这样一来可简化用户进程的实现,把一些共性的、烦琐的、与硬件相关、与特权指令相关的任务放到操作系统层来实现,但提供一个简洁的接口给用户进程调用;二来这层接口事先可规定好,且严格检查用户进程传递进来的参数和操作系统要返回的数据,使得让操作系统给用户进程服务的同时,保护操作系统不会被用户进程破坏。

从硬件层面上看,需要硬件能够支持在用户态的用户进程通过某种机制切换到内核态。实验 1 讲述中断硬件支持和软件处理过程其实就可以用来完成系统调用所需的软硬件支持。下面我们来看看如何在 ucore 中实现系统调用。

1. 初始化系统调用对应的中断描述符

在 ucore 初始化函数 kern_init 中调用了 idt_init 函数来初始化中断描述符表，并设置一个特定中断号的中断门，专门用于用户进程访问系统调用。此事由 ide_init 函数完成。

```
void
idt_init(void) {
    extern uintptr_t __vectors[];
    int i;
    for(i=0; i<sizeof(idt)/sizeof(struct gatedesc); i++) {
        SETGATE(idt[i], 1, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);
    lidt(&idt_pd);
}
```

在上述代码中，可以看到在执行加载中断描述符表 lidt 指令前，专门设置了一个特殊的中断描述符 idt[T_SYSCALL]，它的特权级设置为 DPL_USER，中断向量处理地址在__vectors[T_SYSCALL]处。这样建好这个中断描述符后，一旦用户进程执行“INT T_SYSCALL”后，由于此中断允许用户态进程产生（注意它的特权级设置为 DPL_USER），所以 CPU 就会从用户态切换到内核态，保存相关寄存器，并跳转到__vectors[T_SYSCALL]处开始执行，形成如下执行路径：

```
vector128(vectors.S)-->__alltraps(trapentry.S)-->trap(trap.c)-->trap_dispatch(trap.c)---->
syscall(syscall.c)-
```

在 syscall 中，根据系统调用号来完成不同的系统调用服务。

2. 建立系统调用的用户库准备

在操作系统中初始化好系统调用相关的中断描述符、中断处理起始地址等之后，还需在用户态的应用程序中初始化好相关工作，简化应用程序访问系统调用的复杂性。为此在用户态建立了一个中间层，即简化的 libc 实现，在 user/libs/ulib.[ch]和 user/libs/syscall.[ch]中完成了对访问系统调用的封装。用户态最终的访问系统调用函数是 syscall，实现如下：

```
static inline int
syscall(int num,…) {
    va_list ap;
    va_start(ap,num);
    uint32_t a[MAX_ARGS];
    int i,ret;
    for(i=0;i<MAX_ARGS;i++) {
        a[i]=va_arg(ap, uint32_t);
    }
    va_end(ap);
    asm volatile(
        "int %1;"
```

```
            : "=a" (ret)
            : "i" (T_SYSCALL),
              "a" (num),
              "d" (a[0]),
              "c" (a[1]),
              "b" (a[2]),
              "D" (a[3]),
              "S" (a[4])
            : "cc", "memory");
    return ret;
}
```

从中可以看出，应用程序调用的 exit、fork、wait、getpid 等库函数最终都会调用 syscall 函数，只是调用的参数不同而已，如果看最终的汇编代码会更清楚：

```
⋮
    34:    8b 55 d4          mov     -0x2c(%ebp),%edx
    37:    8b 4d d8          mov     -0x28(%ebp),%ecx
    3a:    8b 5d dc          mov     -0x24(%ebp),%ebx
    3d:    8b 7d e0          mov     -0x20(%ebp),%edi
    40:    8b 75 e4          mov     -0x1c(%ebp),%esi
    43:    8b 45 08          mov     0x8(%ebp),%eax
    46:    cd 80             int     $0x80
    48:    89 45 f0          mov     %eax,-0x10(%ebp)
⋮
```

可以看到其实是把系统调用号放到 EAX，其他 5 个参数 a[0]～a[4]分别保存到 EDX、ECX、EBX、EDI、ESI 寄存器中，最多用 6 个寄存器来传递系统调用的参数，且系统调用的返回结果是 EAX。例如，对于 getpid 库函数而言，系统调用号（SYS_getpid＝18）是保存在 EAX 中，返回值（调用此库函数的当前进程号 pid）也在 EAX 中。

3. 与用户进程相关的系统调用

在本实验中，与进程相关的各个系统调用属性如表 6-1 所示。

表 6-1　与进程相关的各个系统调用属性

系统调用名	含　　义	具体完成服务的函数
SYS_exit	process exit	do_exit
SYS_fork	create child process，dup mm	do_fork－－>wakeup_proc
SYS_wait	wait child process	do_wait
SYS_exec	after fork，process execute a new program	load a program and refresh the mm

续表

系统调用名	含　　义	具体完成服务的函数
SYS_yield	process flag itself need resecheduling	proc－＞need_sched＝1, then scheduler will rescheule this process
SYS_kill	kill process	do_kill－－＞proc－＞flags |＝ PF_EXITING, －－＞wakeup_proc－－＞do_wait－－＞do_exit
SYS_getpid	get the process's pid	

通过这些系统调用,可方便地完成从进程/线程创建到退出的整个运行过程。

4. 系统调用的执行过程

与用户态的函数库调用执行过程相比,系统调用执行过程有四点主要的不同。

(1) 不是通过 CALL 指令而是通过 INT 指令发起调用。

(2) 不是通过 RET 指令,而是通过 IRET 指令完成调用返回。

(3) 当到达内核态后,操作系统需要严格检查系统调用传递的参数,确保不破坏整个系统的安全性。

(4) 执行系统调用可导致进程等待某事件发生,从而可引起进程切换。

下面我们以 getpid 系统调用的执行过程大致看看操作系统是如何完成整个执行过程的。当用户进程调用 getpid 函数,最终执行到"INT T_SYSCALL"指令后,CPU 根据操作系统建立的系统调用中断描述符,转入内核态,并跳转到 vector128 处(kern/trap/vectors.S),开始操作系统的系统调用执行过程,函数调用和返回操作的关系如下所示:

```
vector128(vectors.S)-->__alltraps(trapentry.S)-->trap(trap.c)-->trap_dispatch(trap.c)--
-->syscall(syscall.c)-->sys_getpid(syscall.c)-->…-->__trapret(trapentry.S)
```

在执行 trap 函数前,软件还需进一步保存执行系统调用前的执行现场,即把与用户进程继续执行所需的相关寄存器等当前内容保存到当前进程的中断帧 trapframe 中(在创建进程时,把进程的 trapframe 放在给进程的内核栈分配的空间的顶部)。软件做的工作在 vector128 和__alltraps 的起始部分:

```
vectors.S::vector128起始处:
    pushl $0
    pushl $128
⋮
trapentry.S::__alltraps起始处:
    pushl %ds
    pushl %es
    pushal
⋮
```

自此,用于保存用户态的用户进程执行现场的 trapframe 的内容填写完毕,操作系统可开始完成具体的系统调用服务。在 sys_getpid 函数中,简单地把当前进程的 pid 成员变量作为函数返回值就是一个具体的系统调用服务。完成服务后,操作系统按调用关系的路径原路返回到__alltraps 中。然后操作系统开始根据当前进程的中断帧内容做恢复执行现场

操作。其实就是把 trapframe 的一部分内容保存到寄存器内容。恢复寄存器内容结束后，调整内核堆栈指针到中断帧的 tf_eip 处，这是内核栈的结构如下：

```
/* below here defined by x86 hardware */
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
/* below here only when crossing rings */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
```

这时执行 IRET 指令后，CPU 根据内核栈的情况恢复到用户态，并把 EIP 指向 tf_eip 的值，即“INT T_SYSCALL”后的那条指令。这样整个系统调用就执行完毕了。

至此，实验 5 中的主要工作描述完毕。

6.4 实验报告要求

从网站上下载 lab5.zip 后，解压得到本文档和代码目录 lab5，完成实验中的各个练习。完成代码编写并检查无误后，在对应目录下执行 make handin 任务，即会自动生成 lab5-handin.tar.gz。最后请一定提前或按时提交到网络学堂上。

注意有 lab5 的注释，代码中所有需要完成的地方(Challenge 除外)都有 lab5 和“Your Code”的注释，请在提交时特别注意保持注释，并将“Your Code”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

辅助材料 A “原理”用户进程的特征

1. 从内核线程到用户进程

在实验 4 中设计实现了进程控制块，并实现了内核线程的创建和简单的调度执行。但实验 4 中没有在用户态执行用户进程的管理机制，即无法体现用户进程的地址空间，以及用户进程间地址空间隔离的保护机制，不支持进程执行过程的用户态和核心态之间的切换，且没有用户进程的完整状态变化的生命周期。其实没有实现的原因是内核线程不需要这些功能。那内核线程相对于用户态线程有何特点呢？

其实我们已经在实验 4 中看到了内核线程，内核线程的管理实现相对简单，其特点是直接使用操作系统(比如 ucore)在初始化中建立的内核虚拟内存地址空间，不同的内核线程之间可以通过调度器实现线程间的切换，达到分时使用 CPU 的目的。由于内核虚拟内存空间是一一映射计算机系统的物理空间的，这使得可用空间的大小不会超过物理空间大小，所以操作系统程序员编写内核线程时，需要考虑有限的地址空间，需要保证各个内核线程在执行过程中不会破坏操作系统的正常运行。这样在实现内核线程管理时，不必考虑涉及与进程相关的虚拟内存管理中的缺页处理、按需分页、写时复制、页换入换出等功能。如果在内核线

程执行过程中出现了访存错误异常或内存不够的情况，就认为操作系统出现错误了，操作系统将直接宕机。在 ucore 中，就是调用 panic 函数，进入内核调试监控器 kernel_debug_monitor。

内核线程管理的思想相对简单，但编写内核线程对程序员的要求很高。从理论上讲（理想情况），如果程序员都是能够编写操作系统级别的“高手”，能够勤俭和高效地使用计算机系统中的资源，且这些“高手”都为他人着想，具有奉献精神，在别的应用需要计算机资源的时候，能够从大局出发，从整个系统的执行效率出发，让出自己占用的资源，那这些“高手”编写出来的程序直接作为内核线程运行即可，也就没有用户进程存在的必要了。

现实与理论的差距是巨大的，能编写操作系统的程序员是极少数的，与当前的应用程序员相比，估计大约差了 3～4 个数量级。如果还要求编写操作系统的程序员考虑其他未知程序员的未知需求，那这样的程序员估计可以成为编程界的“上帝”了。

从应用程序编写和运行的角度看，既然程序员都不是“上帝”，操作系统程序员就需要给应用程序员编写的程序提供一个既“宽松”又“严格”的执行环境，让对内存大小和 CPU 使用时间等资源的限制没有仔细考虑的应用程序都能在操作系统中正常运行，且即使程序太可靠，也只能破坏自己，而不能破坏其他运行程序和整个系统。“严格”就是安全性保证，即应用程序执行不会破坏在内存中存在的其他应用程序和操作系统的内存空间等独占的资源；“宽松”就算是方便性支持，即提供给应用程序尽量丰富的服务功能和一个远大于物理内存空间的虚拟地址空间，使得应用程序在执行过程中不必考虑很多烦琐的细节（比如如何初始化 PCI 总线和外设等，如果管理物理内存等）。

2. 让用户进程正常运行的用户环境

在操作系统原理的介绍中，一般提到进程的概念其实主要是指用户进程。从操作系统的设计和实现的角度看，用户进程是指一个应用程序在操作系统提供的一个用户环境中的一次执行过程。这里的重点是用户环境。用户环境有什么功能？用户环境指的是什么？

从功能上看，操作系统提供的这个用户环境有两方面的特点。一方面与存储空间相关，即限制用户进程可以访问的物理地址空间，且让各个用户进程之间的物理内存空间访问不重叠，这样可以保证不同用户进程之间不能相互破坏各自的内存空间，利用虚拟内存的功能（页换入和换出）。给用户进程提供了远大于实际物理内存空间的虚拟内存空间。

另一方面与执行指令相关，即限制用户进程可执行的指令，不能让用户进程执行特权指令（比如修改页表起始地址），从而保证用户进程无法破坏系统。但如果不能执行特权指令，则很多功能（比如访问磁盘等）无法实现，所以需要提供某种机制，让操作系统完成需要特权指令才能做的各种服务功能，给用户进程一个“服务窗口”，用户进程可以通过这个“窗口”向操作系统提出服务请求，由操作系统来帮助用户进程完成需要特权指令才能做的各种服务。另外，还要有一个“中断窗口”，让用户进程不主动放弃使用 CPU 时，操作系统能够通过这个“中断窗口”强制让用户进程放弃使用 CPU，从而让其他用户进程有机会执行。

基于功能分析，我们就可以把这个用户环境定义为如下组成部分。

（1）建立用户虚拟空间的页表和支持页换入和换出机制的用户内存访存错误异常服务例程：提供地址隔离和超过物理空间大小的虚存空间。

（2）应用程序执行的用户态 CPU 特权级：在用户态 CPU 特权级，应用程序只能执行一般指令，如果是特权指令，结果不是无效就是产生“执行非法指令”异常。

（3）系统调用机制：给用户进程提供“服务窗口”。

(4) 中断响应机制：给用户进程设置“中断窗口”，这样产生中断后，当前执行的用户进程将被强制打断，CPU 控制权将被操作系统的中断服务例程使用。

3. 用户态进程的执行过程分析

在这个环境下运行的进程就是用户进程。如果用户进程由于某种原因进入内核态后，那么在内核态执行的是什么呢？还是用户进程吗？首先分析一下用户进程怎样会进入内核态呢？回顾一下 lab1，就可以知道当产生外设中断、CPU 执行异常（比如访存错误）、陷入（系统调用），用户进程就会切换到内核中的操作系统中。表面上看，到内核态后，操作系统取得了 CPU 的控制权，所以现在执行的应该是操作系统代码，由于此时 CPU 处于核心态特权级，所以操作系统的执行过程就就应该是内核进程了。这样理解忽略了操作系统的具体实现。如果考虑操作系统的具体实现，应该如何来理解进程呢？

从进程控制块的角度看，如果执行了进程执行现场（上下文）的切换，就认为到另外一个进程执行了，并且进程的分界点设定在执行进程切换的前后。到底切换了什么呢？其实只是切换了进程的页表和相关硬件寄存器，这些信息都保存在进程控制块中的相关域中。所以，我们可以把执行应用程序的代码一直到执行操作系统中的进程切换处为止都认为是一个应用程序的执行过程（其中有操作系统的部分代码执行过程）即进程。因为在这个过程中，没有更换到另外一个进程控制块的进程的页表和相关硬件寄存器。

从指令执行的角度看，如果再仔细分析一下操作系统这个软件的特点并细化一下进入内核的原因，就可以看出进一步进行划分。操作系统的主要功能是给上层应用提供服务，管理整个计算机系统中的资源。所以操作系统虽然是一个软件，但其实是一个基于事件的软件，这里操作系统需要响应的事件包括三类：外设中断、CPU 执行异常（比如访存错误）、陷入（系统调用）。如果用户进程通过系统调用要求操作系统提供服务，那么从用户进程的角度看，操作系统就是一个特殊的软件库（比如相对于用户态的 libc 库，操作系统可看做内核态的 libc 库），完成用户进程的需求，从执行逻辑上看，是用户进程“主观”执行的一部分，即用户进程“知道”操作系统要做的事情。那么在这种情况下，进程的代码空间包括用户态的执行程序和内核态响应用户进程通过系统调用而在核心特权态执行服务请求的操作系统代码，这种情况下的进程的内存虚拟空间也包括两部分：用户态的虚地址空间和核心态的虚地址空间。但如果此时发生的事件是外设中断和 CPU 执行异常，虽然 CPU 控制权也转入到操作系统中的中断服务例程，但这些内核执行代码执行过程是用户进程“不知道”的，是另外一段执行逻辑。那么在这种情况下，实际上是执行了两段目标不同的执行程序，一个是代表应用程序的用户进程，一个是代表中断服务例程处理外设中断和 CPU 执行异常的内核线程。这个用户进程和内核线程在产生中断或异常的时候，CPU 硬件就完成了它们之间的指令流切换。

4. 用户进程的运行状态分析

用户进程在其执行过程中会存在很多种不同的执行状态，根据操作系统的原理，一个用户进程一般的运行状态有五种：创建（new）态、就绪（ready）态、运行（running）态、等待（blocked）态、退出（exit）态。各个状态之间会由于发生了某事件而进行状态转换。

但在用户进程的执行过程中，具体在哪个时间段处于上述状态呢？上述状态是如何转变的呢？首先，我们看创建态，操作系统完成进程的创建工作，而体现进程存在的就是进程控制块，所以一旦操作系统创建了进程控制块，则可以认为此时进程就已经存在了，但由于

进程能够运行的各种资源还没准备好，所以此时的进程处于创建态。创建了进程控制块后，进程并不能执行，还需准备好各种资源，如果把进程执行所需要的虚拟内存空间、执行代码、要处理的数据等都准备好了，则此时进程可以执行，但还没有被操作系统调度，需要等待操作系统选择这个进程执行，于是把这个做好“执行准备”的进程放入到一个队列中，并可以认为此时进程处于就绪态。当操作系统的调度器从就绪进程队列中选择了一个就绪进程后，通过执行进程切换，就让这个被选上的就绪进程执行，此时进程就处于运行态。到了运行态后，会出现三种事件。如果进程需要等待某个事件（比如主动睡眠 10s，或进程访问某个内存空间，但此内存空间被换出到硬盘 swap 分区中，进程不得不等待操作系统把缓慢的硬盘上的数据重新读回到内存中），那么操作系统会把 CPU 给其他进程执行，并把进程状态从运行态转换为等待态。如果用户进程的应用程序逻辑流程执行结束了，那么操作系统会把 CPU 给其他进程执行，并把进程状态从运行态转换为退出态，并准备回收用户进程占用的各种资源，当把表示整个进程存在的进程控制块也回收了，这进程就不存在了。在这整个回收过程中，进程都处于退出态。考虑到在内存中存在多个处于就绪态的用户进程，但只有一个 CPU，所以为了公平起见，每个就绪态进程都只有有限的时间片，当一个运行态的进程用完它的时间片后，操作系统会剥夺此进程的 CPU 使用权，并把此进程状态从运行态转换为就绪态，最后把 CPU 给其他进程执行。如果某个处于等待态的进程所等待的事件产生了（比如睡眠时间到，或需要访问的数据已经从硬盘换入到内存中），则操作系统会通过把等待此事件的进程状态从等待态转到就绪态。这样进程的整个状态转换形成了一个有限状态自动机。

第 7 章　实验 6：调度器

7.1　实验目的

（1）理解操作系统的调度管理机制。

（2）熟悉 ucore 的系统调度器框架，以及默认的 Round-Robin 调度算法。

（3）基于调度器框架实现一个（Stride Scheduling）调度算法来替换默认的调度算法。

7.2　实验内容

实验 5 完成了用户进程的管理，可在用户态运行多个进程。但到目前为止，采用的调度策略是很简单的 FIFO 调度策略。本次实验主要是熟悉 ucore 的系统调度器框架，以及基于此框架的 Round-Robin（RR）调度算法。然后参考 RR 调度算法的实现，完成 Stride Scheduling 调度算法。

7.2.1　练习

练习 0：填写已有实验。

本实验依赖实验 1～实验 5。请把已做的实验 2～实验 5 的代码填入本实验中代码中有 lab1、lab2、lab3、lab4、lab5 的注释相应部分，并确保编译通过。注意：为了能够正确执行 lab6 的测试应用程序，可能需对已完成的实验～实验 5 的代码进行进一步改进。

练习 1：使用 Round-Robin 调度算法（不需要编码）。

完成练习 0 后，建议大家比较一下（可用 kdiff3 等文件比较软件）个人完成的 lab5 和练习 0 完成后的刚修改的 lab6 之间的区别，分析了解 lab6 采用 RR 调度算法后的执行过程。执行 make grade，大部分测试用例应该通过。但执行 priority. c 应该过不去。

练习 2：实现 Stride Scheduling 调度算法（需要编码）。

首先需要换掉 RR 调度器的实现，即用 default_sched_stride_c 覆盖 default_sched. c。然后根据此文件和后续文档对 Stride 度器的相关描述，完成 Stride 调度算法的实现。

后面的实验文档部分给出了 Stride 调度算法的大体描述。这里给出 Stride 调度算法的一些相关的资料（目前网上中文的资料比较欠缺）。

（1）http://wwwagss. informatik. uni-kl. de/Projekte/Squirrel/stride/node3. html。

（2）http://citeseerx. ist. psu. edu/viewdoc/summary? doi=10. 1. 1. 138. 3502&rank=1。

（3）也可通过 Google 搜索“Stride Scheduling”来查找相关资料。

执行：make grade 。如果所显示的应用程序检测都输出 ok，则基本正确。如果只是 priority. c 过不去，可执行 make run-priority 命令来单独调试它。大致执行结果可看本章附录（使用的是 qemu-1. 0. 1）。

扩展练习 Challenge：实现 Linux 的 CFS 调度算法。

在 ucore 的调度器框架下实现下 Linux 的 CFS 调度算法。可阅读相关 Linux 内核书籍或查询网上资料，可了解 CFS 的细节，然后大致实现在 ucore 中。

7.2.2 项目组成

目录文件结构图如图 7-1 所示。

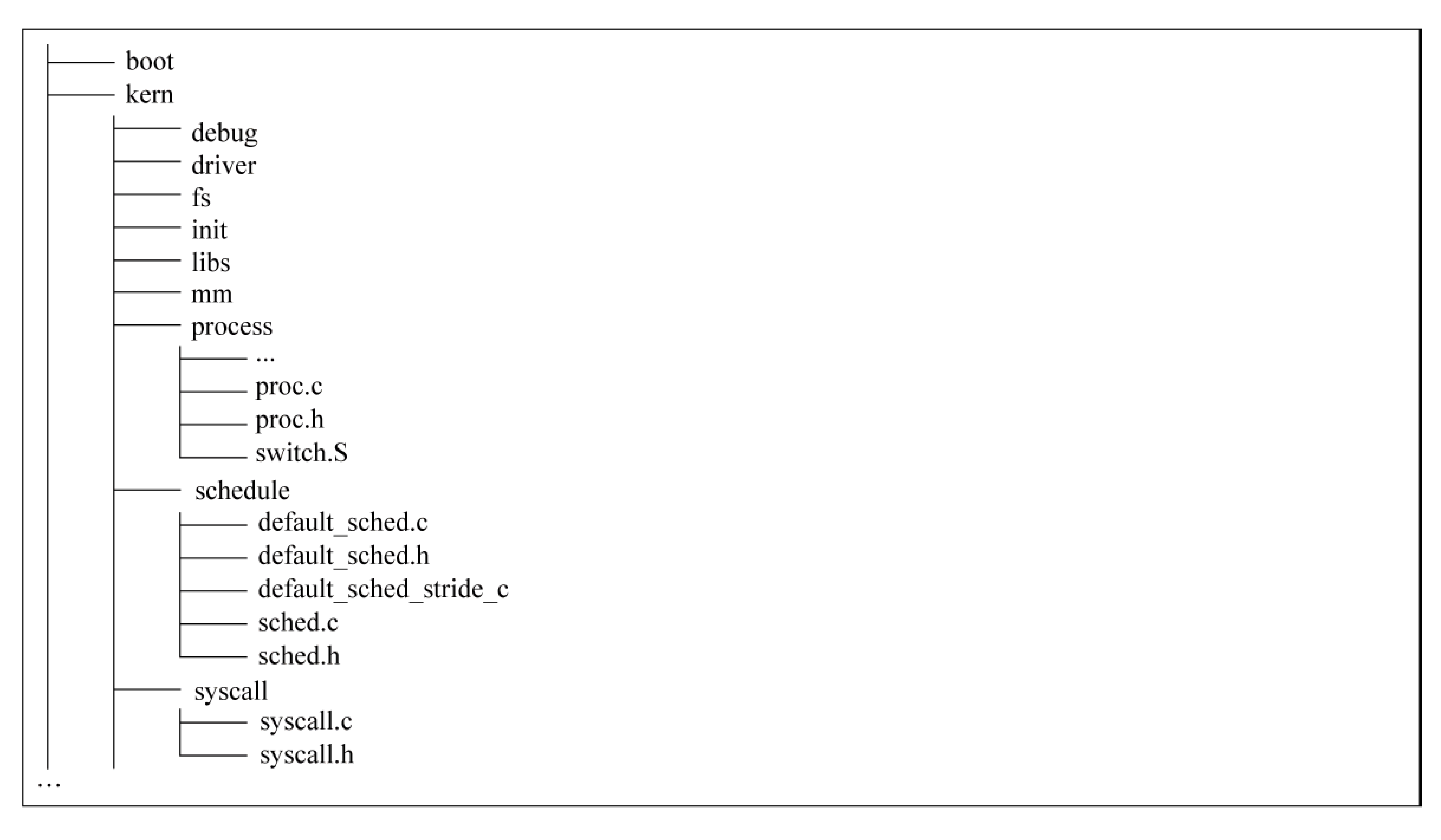


图 7-1 目录文件结构图

相对与实验 5，实验 6 主要增加的文件有 default_sched. c、default_sched. h、default_sched_stricle_c 等，主要修改的文件有 proc. c、proc. h、sched. c、sched. h、syscall. c 和 syscall. h。主要改动如下。

（1）libs/skew_heap. h：提供了基本的优先队列数据结构，为本次实验提供了抽象数据结构方面的支持。

（2）kern/process/proc. [ch]：proc. h 中扩展了 proc_struct 的成员变量，用于 RR 和 stride 调度算法。proc. c 中实现了 lab6_set_priority，用于设置进程的优先级。

（3）kern/schedule/{sched. h，sched. c}：定义了 ucore 的调度器框架，其中包括相关的数据结构（包括调度器的接口和运行队列的结构）和具体的运行时机制。

（4）kern/schedule/{default_sched. h，default_sched. c}：具体的 Round-robin 算法，在本次实验中需要了解其实现。

（5）kern/schedule/default_sched_stride_c：Stride Scheduling 调度器的基本框架，在此次实验中需要填充其中的空白部分以实现一个完整的 Stride 调度器。

（6）kern/syscall/syscall. [ch]：增加了 sys_gettime 系统调用，便于用户进程获取当前时钟值；增加了 sys_lab6_set_priority 系统调用，便于用户进程设置进程优先级（给 priority. c 用）。

（7）user/{matrix. c，priority. c，…}：相关的一些测试用户程序、测试调度算法的正确

性，user 目录下包含但不限于这些程序。在完成实验过程中，建议阅读这些测试程序，以了解这些程序的行为，便于进行调试。

7.3 调度框架和调度算法设计与实现

7.3.1 实验执行流程概述

在实验 5 中创建了用户进程，并让它们正确运行。这中间也实现了 FIFO 调度策略。可通过阅读实验 5 下的 kern/schedule/sched.c 的 schedule 函数的实现来了解其 FIFO 调度策略。与实验 5 相比，实验 6 专门需要针对处理器调度框架和各种算法进行设计与实现，为此对 ucore 的调度部分进行了适当的修改，使得 kern/schedule/sched.c 只实现调度器框架，不再涉及具体的调度算法实现，而调度算法在单独的文件（default_sched.[ch]）中实现。

除此之外，实验中还涉及了 idle 进程的概念。当 CPU 没有进程可以执行的时候，系统应该如何工作？在实验 5 的 scheduler 实现中，ucore 内核不断地遍历进程池，直到找到第一个 runnable 状态的 process，调用并执行它。也就是说，当系统没有进程可以执行的时候，它会把所有 CPU 时间用在搜索进程池，以实现 idle 的目的。但是这样的设计不被大多数操作系统所采用，原因在于它将进程调度和 idle 进程两种不同的概念混在了一起，而且，当调度器比较复杂时，schedule 函数本身也会比较复杂，这样的设计结构很不清晰而且难免会出现错误。所以在此次实验中，ucore 建立了一个单独的进程（kern/process/proc.c 中的 idleproc）作为 CPU 空闲时的 idle 进程，这个程序是通常一个死循环。本实验需要了解这个程序的实现。

接下来可看看实验 6 的大致执行过程，在 init.c 中的 kern_init 函数增加了对 sched_init 函数的调用。sched_init 函数主要完成了对实现特定调度算法的调度类（sched_class）的绑定，使得 ucore 在后续的执行中，能够通过调度框架找到实现特定调度算法的调度类并完成进程调度相关工作。为了更好地理解实验 6 的整个运行过程，这里需要关注的重点问题如下。

（1）何时或何事件发生后需要调度？

（2）何时或何事件发生后需要调整实现调度算法所涉及的参数？

（3）如何基于调度框架设计具体的调度算法？

（4）如何灵活应用链表等数据结构管理进程调度？

大家可带着这些问题进一步阅读后续的内容。

7.3.2 计时器的原理和实现

在传统的操作系统中，计时器是其中一个基础而重要的功能，它提供了基于时间事件的调度机制。在 ucore 中，timer 中断（irq0）给操作系统提供了有一定间隔的时间事件，操作系统将其作为基本的调度和计时单位（记两次时间中断之间的时间间隔为一个时间片，timer splice）。

基于此时间单位，操作系统得以向上提供基于时间点的事件，并实现基于时间长度的等待和唤醒机制。在每个时钟中断发生时，操作系统产生对应的时间事件。应用程序或者操

作系统的其他组件可以以此来构建更复杂和高级的调度。

sched.h、sched.c 定义了有关 timer 的各种相关接口来使用 timer 服务，其中主要包括如下。

（1）typedef struct{…}timer_t：定义了 timer_t 的基本结构，其可以用 sched.h 中的 timer_init 函数对其进行初始化。

（2）void timer_init(timer t * timer, struct proc_struct * proc, int expires)：对某计时器进行初始化，让它在 expires 时间片之后唤醒 proc 进程。

（3）void add_timer(timer t * timer)：向系统添加某个初始化过的 timer_t，该计时器在指定时间后被激活，并将对应的进程唤醒至 runnable（如果当前进程处在等待状态）。

（4）void del_timer(timer_t * time)：向系统删除（或者说取消）某一个计时器。该计时器在取消后不会被系统激活并唤醒进程。

（5）void run_timer_list(void)：更新当前系统时间点，遍历当前所有处在系统管理内的计时器，找出所有应该激活的计数器，并激活它们。该过程只在每次计时器中断时被调用。在 ucore 中，其还会调用调度器事件处理程序。

一个 timer_t 在系统中的存活周期可以被描述如下。

① timer_t 在某个位置被创建和初始化，并通过 add_timer 加入系统管理列表中。

② 系统时间被不断累加，直到 run_timer_list 发现该 timer_t 到期。

③ run_timer_list 更改对应的进程状态，并从系统管理列表中移除该 timer_t。尽管本次实验并不需要填充计时器相关的代码，但是作为系统重要的组件（同时计时器也是调度器的一部分），本实验要求了解其相关机制和在 ucore 中的实现方法。接下来的实验描述将会在一定程度上忽略计时器对调度带来的影响，即不考虑基于固定时间点的调度。

7.3.3 进程状态

在此次实验中，进程状态之间的转换需要有一个更为清晰的表述，在 ucore 中，runnable 的进程会被放在运行队列中。值得注意的是，在具体实现中，ucore 定义的进程控制块 struct proc_struct 包含了成员变量 state，用于描述进程的运行状态，而 running 和 runnable 共享同一个状态（state）值（PROC_RUNNABLE）。不同之处在于处于 running 态的进程不会放在运行队列中。进程的正常生命周期如下。

（1）进程首先在 CPU 初始化或者 sys_fork 的时候被创建，当为该进程分配了一个进程描述符之后，该进程进入 uninit 态（在 proc.c 中的 alloc_proc）。

（2）当进程完全完成初始化之后，该进程转为 runnable 态。

（3）当到达调度点时，由调度器 sched_class 根据运行队列 rq 的内容来判断一个进程是否应该被运行，即把处于 runnable 态的进程转换成 running 状态，从而占用 CPU 执行。

（4）running 态的进程通过 wait 等系统调用被阻塞，进入 sleeping 态。

（5）sleeping 态的进程被 wakeup 变成 runnable 态的进程。

（6）running 态的进程主动 exit 变成 zombie 态，然后由其父进程完成对其资源的最后释放，子进程的进程控制块成为 unused。

（7）所有从 runnable 态变成其他状态的进程都要出运行队列，反之，被放入某个运行队列中。

7.3.4 进程调度实现

1. 内核抢占点

调度本质上体现了对 CPU 资源的抢占。对于用户进程而言,由于有中断的产生,可以随时打断用户进程的执行,转到操作系统内部,从而给了操作系统以调度控制权,让操作系统可以根据具体情况(比如用户进程时间片已经用完了)选择其他用户进程执行。这体现了用户进程的可抢占性(preemptive)。但如果把 ucore 操作系统也看成一个特殊的内核进程或多个内核线程的集合,那么 ucore 是否也是可抢占的呢?其实 ucore 内核执行是不可抢占的(non-preemptive),即在执行“任意”内核代码时,CPU 控制权可被强制剥夺。这里需要注意,不是在所有情况下 ucore 内核执行都是不可抢占的,有以下几种“固定”情况是例外。

(1) 进行同步互斥操作,比如争抢一个信号量、锁(lab7 中会详细分析)。

(2) 进行磁盘读写等耗时的异步操作,由于等待完成的耗时太长,ucore 会调用 shcedule 让其他就绪进程执行。

这几种情况其实都是由于当前进程所需的某个资源(也可称为事件)无法得到满足,无法继续执行下去,从而不得不主动放弃对 CPU 的控制权。如果参照用户进程任何位置都可被内核打断并放弃 CPU 控制权的情况,这些在内核中放弃 CPU 控制权的执行地点是“固定”而不是“任意”的,不能体现内核任意位置都可抢占性的特点。搜寻一下实验 5 的代码,可发现在如下几处地方调用了 shedule 函数,如表 7-1 所示。

表 7-1 调用进程调度函数 schedule 的位置和原因

编号	位 置	原 因
1	proc.c::do_exit	用户线程执行结束,主动放弃 CPU 控制权
2	proc.c::do_wait	用户线程等待子进程结束,主动放弃 CPU 控制权
3	proc.c::init_main	(1) initproc 内核线程等待所有用户进程结束,如果没有结束,就主动放弃 CPU 控制权 (2) initproc 内核线程在所有用户进程结束后,让 kswapd 内核线程执行 10 次,用于回收空闲内存资源
4	proc.c::cpu_idle	idleproc 内核线程的工作就是等待有处于就绪态的进程或线程,如果有就调用 schedule 函数
5	sync.h::lock	在获取锁的过程中,如果无法得到锁,则主动放弃 CPU 控制权
6	trap.c::trap	如果当前进程在用户态被打断,且当前进程控制块的成员变量 need_resched 设置为 1,则当前线程会放弃 CPU 控制权

仔细分析上述位置,第 1、2、5 处的执行位置体现了由于获取某种资源一时等不到满足、进程要退出、进程要睡眠等原因而不得不主动放弃 CPU。第 3、4 处的执行位置比较特殊,initproc 内核线程等待用户进程结束而执行 schedule 函数;idle 内核线程在没有进程处于就绪态时才执行,一旦有了就绪态的进程,它将执行 schedule 函数完成进程调度。这里只有第 6 处的位置比较特殊:

```
if(!in_kernel) {
    ⋮
```

```
    if (current->need_resched) {
        schedule();
    }
}
```

这里表明了只有当进程在用户态执行到“任意”某处用户代码位置时发生了中断，且当前进程控制块成员变量 need_resched 为 1(表示需要调度了)时，才会执行 shedule 函数。这实际上体现了对用户进程的可抢占性。如果没有第一行的 if 语句，那么就可以体现对内核代码的可抢占性。但如果要把这一行 if 语句去掉，就不得不实现对 ucore 中的所有全局变量的互斥访问操作，以防止 race condition 现象，这样 ucore 的实现复杂度会增加不少。

2. 进程切换过程

进程调度函数 schedule 选择了下一个将占用 CPU 执行的进程后，将调用进程切换，从而让新的进程得以执行。通过实验 4 和实验 5 的理解，应该已经对进程调度和上下文切换有了初步的认识。在实验 5 中，结合调度器框架的设计，可对 ucore 中的进程切换以及堆栈的维护和使用等有更加深刻的认识。假定有两个用户进程，在两者进行进程切换的过程中，具体的步骤如下。

首先在执行某进程 A 的用户代码时，出现了一个 trap (例如，是一个外设产生的中断)，这时就会从进程 A 的用户态切换到内核态(过程(1))，并且保存好进程 A 的 trapframe；当内核态处理中断时发现需要进行进程切换时，ucore 要通过 schedule 函数选择下一个将占用 CPU 执行的进程(即进程 B)，然后会调用 proc_run 函数，proc_run 函数进一步调用 switch_to 函数，切换到进程 B 的内核态(过程(2))，继续进程 B 上一次在内核态的操作，并通过 iret 指令，最终将执行权转交给进程 B 的用户空间(过程(3))。

当进程 B 由于某种原因发生中断之后(过程(4))，会从进程 B 的用户态切换到内核态，并且保存好进程 B 的 trapframe；当内核态处理中断时发现需要进行进程切换时，即需要切换到进程 A，ucore 再次切换到进程 A(过程(5))，会执行进程 A 上一次在内核调用 schedule(具体还要跟踪到 switch_to 函数)函数返回后的下一行代码，这行代码当然还是在进程 A 的上一次中断处理流程中。最后当进程 A 的中断处理完毕的时候，执行权又会反交给进程 A 的用户代码(过程(6))。这就是在只有两个进程的情况下，进程切换间的大体流程。

需要强调的几点如下。

(1) 需要透彻理解在进程切换以后，程序是从哪里开始执行的？需要注意到虽然指令还是同一个 CPU 上执行，但是此时已经是另外一个进程在执行了，且使用的资源已经完全不同了。

(2) 内核在第一个程序运行的时候，需要进行哪些操作？有了实验 4 和实验 5 的经验，可以确定，内核启动第一个用户进程的过程，实际上是从进程启动时的内核状态切换到该用户进程的内核状态的过程，而且该用户进程在用户态的起始入口应该是 forkret。

7.3.5 调度框架和调度算法

1. 设计思路

实行一个进程调度策略，到底需要实现哪些基本功能对应的数据结构？首先考虑到一

个无论哪种调度算法都需要选择一个就绪进程来占用CPU运行。为此我们可把就绪进程组织起来，可用队列(双向链表)、二叉树、红黑树、数组等不同的组织方式。

在操作方面，如果需要选择一个就绪进程，就可以从基于某种组织方式的就绪进程集合中选择出一个进程执行。需要注意，这里“选择”和“出”是两个操作，选择是在集合中挑选一个“合适”的进程，“出”意味着离开就绪进程集合。另外考虑到一个处于运行态的进程还会由于某种原因(比如时间片用完了)回到就绪态而不能继续占用CPU执行，这就会重新进入到就绪进程集合中。这两种情况就形成了调度器相关的三个基本操作：在就绪进程集合中选择、进入就绪进程集合和离开就绪进程集合。这三个操作属于调度器的基本操作。

在进程的执行过程中，就绪进程的等待时间和执行进程的执行时间是影响调度选择的重要因素，这两个因素随着时间的流逝和各种事件的发生在不停地变化，比如处于就绪态的进程等待调度的时间在增长，处于运行态的进程所消耗的时间片在减少等。这些进程状态变化的情况需要及时让进程调度器知道，便于选择更合适的进程执行。所以这种进程变化的情况就形成了调度器相关的一个变化感知操作：timer时间事件感知操作。这样在进程运行或等待的过程中，调度器可以调整进程控制块中与进程调度相关的属性值(比如消耗的时间片、进程优先级等)，并可能导致对进程组织形式的调整(比如以时间片大小的顺序来重排双向链表等)，并最终可能导致调选择新的进程占用CPU运行。这个操作属于调度器的进程调度属性调整操作。

2. 数据结构

在理解框架之前，需要先了解一下调度器框架所需要的数据结构。

(1) 通常的操作系统中，进程池是很大的(虽然在ucore中，MAX_PROCESS很小)。在ucore中，调度器引入run-queue(简称rq，即运行队列)的概念，通过链表结构管理进程。

(2) 由于目前ucore设计运行在单CPU上，其内部只有一个全局的运行队列，用来管理系统内全部的进程。

(3) 运行队列通过链表的形式进行组织。链表的每一个节点是一个list_entry_t，每个list_entry_t又对应到了struct proc_struct *，这其间的转换是通过宏le2proc来完成的。具体来说，我们知道在struct proc_struct中有一个叫run_link的list_entry_t，因此可以通过偏移量逆向找到对因某个run_list的struct proc_struct。即进程结构指针proc＝le2proc(链表节点指针，run_link)。

(4) 为了保证调度器接口的通用性，ucore调度框架定义了如下接口，该接口中，几乎全部成员变量均为函数指针。具体的功能会在后面的框架说明中介绍。

```
struct sched_class {
//调度器的名字
const char* name;
//初始化运行队列
void(* init)(struct run_queue* rq);
//将进程 p插入队列 rq
void(* enqueue)(struct run_queue* rq, struct proc_struct* p);
//将进程 p从队列 rq中删除
void(* dequeue)(struct run_queue* rq, struct proc_struct* p);
 //返回运行队列中下一个可执行的进程
```

```
struct proc_struct * (* pick_next) (struct run_queue * rq);
//timetick处理函数
void(* proc_tick) (struct  run_queue * rq,struct proc_struct * p);
};
```

此外,proc. h 中的 struct proc_struct 中也记录了一些调度相关的信息:

```
struct proc_struct {
⋮
//该进程是否需要调度,只对当前进程有效
volatile bool need_resched;
//该进程的调度链表结构,该结构内部的链接组成了运行队列列表
list_entry_t run_link;
//该进程剩余的时间片,只对当前进程有效
int time_slice;
//round-robin调度器并不会用到以下成员
//该进程在优先队列中的节点,仅在 lab6中使用
skew_heap_entry_t  lab6_run_pool;
//该进程的调度优先级,仅在 lab6中使用
uint32_t lab6_priority;
//该进程的调度步进值,仅在 lab6中使用
uint32_t lab6_stride;
};
```

在此次实验中,需要了解 default_sched. c 中的实现 RR 调度算法的函数。在该文件中,可以看到 ucore 已经为 RR 调度算法创建好了一个名为 RR_sched_class 的调度策略类。

通过数据结构 struct run_queue 来描述完整的 run_queue(运行队列)。它的主要结构如下:

```
struct run_queue {
//其运行队列的哨兵结构,可以看做队列头和尾
list_entry_t run_list;
//优先队列形式的进程容器,只在 lab6中使用
skew_heap_entry_t * lab6_run_pool;
//表示其内部的进程总数
unsigned int proc_num;
//每个进程一轮占用的最多时间片
int max_time_slice;
};
```

在 ucore 框架中,运行队列存储的是当前可以调度的进程,所以,只有状态为 runnable 的进程才能够进入运行队列。当前正在运行的进程并不会在运行队列中,这一点需要注意。

3. 调度点的相关关键函数

虽然进程各种状态变化的原因和导致的调度处理各异,但其实仔细观察各个流程的共性部分,会发现其中只涉及了三个关键调度相关函数:wakup_proc、shedule、run_timer_list。如果我们能够让这三个调度相关函数的实现与具体调度算法无关,那么就可以认为

ucore 实现了一个与调度算法无关的调度框架。

wakeup_proc 函数其实完成了把一个就绪进程放入就绪进程队列中的工作，为此还调用了一个调度类接口函数 sched_class_enqueue，这使得 wakeup_proc 函数的实现与具体调度算法无关。schedule 函数完成了与调度框架和调度算法相关的三件事情：把当前继续占用 CPU 执行的运行进程放入到就绪进程队列中，从就绪进程队列中选择一个“合适”就绪进程，把这个“合适”的就绪进程从就绪进程队列中摘除。通过调用三个调度类接口函数 sched_class_enqueue、sched_class_pick_next、sched_class_enqueue 来使得完成这三件事情与具体的调度算法无关。run_timer_list 函数在每次 timer 中断处理过程中被调用，从而可用来调用调度算法所需的 timer 时间事件感知操作，调整相关进程的进程调度相关的属性值。通过调用调度类接口函数 sched_class_proc_tick 使得此操作与具体调度算法无关。

这里涉及了一系列调度类接口函数。

（1）sched_class_enqueue。

（2）sched_class_dequeue。

（3）sched_class_pick_next。

（4）sched_class_proc_tick。

这 4 个函数的实现其实就是调用某基于 sched_class 数据结构的特定调度算法实现的 4 个指针函数。采用这样的调度类框架后，如果我们需要实现一个新的调度算法，则需要定义一个针对此算法的调度类的实例，一个就绪进程队列的组织结构描述就行了，其他的事情都可交给调度类框架来完成。

4. RR 调度算法的实现

RR 调度算法的调度思想是让所有 runnable 态的进程分时轮流使用 CPU 时间。RR 调度器维护当前 runnable 进程的有序运行队列。当前进程的时间片用完之后，调度器将当前进程放置到运行队列的尾部，再从其头部取出进程进行调度。RR 调度算法的就绪队列在组织结构上也是一个双向链表，只是增加了一个成员变量，表明在此就绪进程队列中的最大执行时间片。而且在进程控制块 proc_struct 中增加了一个成员变量 time_slice，用来记录进程当前的可运行时间片。这是由于 RR 调度算法需要考虑执行进程的运行时间不能太长。在每个 timer 到的时候，操作系统会递减当前执行进程的 time_slice，当 time_slice 为 0 时，就意味着这个进程运行了一段时间（这个时间片称为进程的时间片），需要把 CPU 让给其他进程执行，于是操作系统就需要让此进程重新回到 rq 的队列尾，且重置此进程的时间片为就绪队列的成员变量最大时间片 max_time_slice 值，然后再从 rq 的队列头取出一个新的进程执行。下面来分析一下其调度算法的实现。

RR_enqueue 的函数实现如下所示，即把某进程的进程控制块指针放入 rq 队列末尾，且如果进程控制块的时间片为 0，则需要把它重置为 rq 成员变量 max_time_slice。这表示如果进程在当前的执行时间片已经用完，需要等到下一次有机会运行时，才能再执行一段时间。

```
static void
RR_enqueue(struct run_queue * rq, struct proc_struct * proc) {
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link));
```

```
        if (proc->time_slice==0||proc->time_slice>rq->max_time_slice) {
            proc->time_slice=rq->max_time_slice;
        }
        proc->rq=rq;
        rq->proc_num++;
    }
```

RR_pick_next 的函数实现如下所示，即选取就绪进程队列 rq 中的队头队列元素，并把队列元素转换成进程控制块指针。

```
    static struct proc_struct *
    FCFS_pick_next(struct run_queue * rq) {
        list_entry_t * le=list_next(&(rq->run_list));
        if(le!=&(rq->run_list)) {
            return le2proc(le,run_link);
        }
        return NULL;
    }
```

RR_dequeue 的函数实现如下所示，即把就绪进程队列 rq 的进程控制块指针的队列元素删除，并把表示就绪进程个数的 proc_num 减 1。

```
    static void
    FCFS_dequeue(struct run_queue * rq,struct proc_struct * proc) {
        assert(!list_empty(&(proc->run_link))&&proc->rq==rq);
        list_del_init(&(proc->run_link));
        rq->proc_num--;
    }
```

RR_proc_tick 的函数实现如下所示，即每次 timer 到时后，trap 函数将会间接调用此函数来把当前执行进程的时间片 time_slice 减 1。如果 time_slice 降到零，则设置此进程成员变量 need_resched 标识为 1，这样在下一次中断来后执行 trap 函数时，会由于当前进程成员变量 need_resched 标识为 1 而执行 schedule 函数，从而把当前执行进程放回就绪队列末尾，而从就绪队列头取出在就绪队列上等待时间最久的那个就绪进程执行。

```
    static void
    RR_proc_tick(struct run_queue * rq,struct proc_struct * proc) {
        if(proc->time_slice>0) {
            proc->time_slice--;
        }
        if(proc->time_slice==0) {
            proc->need_resched=1;
        }
    }
```

7.3.6 Stride Scheduling

1. 基本思路

提示：请先看练习 2 中提到的论文。理解后再看下面的内容。

考查 round-robin 调度器，在假设所有进程都充分使用了其拥有的 CPU 时间资源的情况下，所有进程得到的 CPU 时间应该是相等的。但是有时候我们希望调度器能够更智能地为每个进程分配合理的 CPU 资源。假设为不同的进程分配不同的优先级，则我们有可能希望每个进程得到的时间资源与它们的优先级成正比关系。Stride 调度是基于这种想法的一个较为典型和简单的算法。除了简单易于实现以外，它还有如下特点。

(1) 可控性：如我们之前所希望的，可以证明 Stride Scheduling 对进程的调度次数正比于其优先级。

(2) 确定性：在不考虑计时器事件的情况下，整个调度机制都是可预知和重现的。该算法的基本思想可以考虑如下。

① 为每个 runnable 的进程设置一个当前状态 stride，表示该进程当前的调度权。另外定义其对应的 pass 值，表示对应进程在调度后，stride 需要进行的累加值。

② 每次需要调度时，从当前 runnable 态的进程中选择 stride 最小的进程调度。

③ 对于获得调度的进程 P，将对应的 stride 加上其对应的步长 pass(只与进程的优先权有关系)。

④ 在一段固定的时间之后，回到步骤②，重新调度当前 stride 最小的进程。可以证明，如果令

```
P.pass= BigStride/P.priority
```

其中，P. priority 表示进程的优先权(大于 1)，而 BigStride 表示一个预先定义的大常数，则该调度方案为每个进程分配的时间将与其优先级成正比。证明过程在这里略去，有兴趣的同学可以在网上查找相关资料。将该调度器应用到 ucore 的调度器框架中，则需要将调度器接口实现如下。

a. init。

初始化调度器类的信息(如果有的话)。

初始化当前的运行队列为一个空的容器结构(比如和 RR 调度算法一样，初始化为一个有序列表)。

b. enqueue。

初始化刚进入运行队列的进程 proc 的 stride 属性。

将 proc 插入放入运行队列中(注意：这里并不要求放置在队列头部)。

c. dequeue。

从运行队列中删除相应的元素。

d. pick next。

扫描整个运行队列，返回其中 stride 值最小的对应进程。

更新对应进程的 stride 值，即 pass＝BIG_STRIDE/P－>priority；P－>stride＋＝pass。

e. proc tick。

检测当前进程是否已用完分配的时间片。如果时间片用完，应该正确设置进程结构的相关标记来引起进程切换。

一个 process 最多可以连续运行 rq. max_time_slice 个时间片。

在具体实现时，有一个需要注意的地方，即 stride 属性的溢出问题，在之前的实现里面

我们并没有考虑 stride 的数值范围，而这个值在理论上是不断增加的，在 stride 溢出以后，基于 stride 的比较可能会出现错误。例如，假设当前存在两个进程 A 和 B，stride 属性采用 16 位无符号整数进行存储。当前队列中元素如下（假设当前运行的进程已经被重新放置进运行队列中）：

A. stride（实际值）	A. stride（理论值）	A. pass $\left(=\frac{\text{BigStride}}{\text{A. priority}}\right)$
65534	65534	100
B. stride（实际值）	B. stride（理论值）	B. pass $\left(=\frac{\text{BigStride}}{\text{B. priority}}\right)$
65535	65535	50

此时应该选择 A 作为调度的进程，而在一轮调度后，队列将如下：

A. stride（实际值）	A. stride（理论值）	A. pass $\left(=\frac{\text{BigStride}}{\text{A. priority}}\right)$
98	65634	100
B. stride（实际值）	B. stride（理论值）	B. pass $\left(=\frac{\text{BigStride}}{\text{B. priority}}\right)$
65535	65535	50

可以看到由于溢出的出现，进程间 stride 的理论比较和实际比较结果出现了偏差。我们首先在理论上分析这个问题：令 PASS_MAX 为当前所有进程里最大的步进值，则可以证明如下结论：对每次 Stride 调度器的调度步骤中，有其最大的步进值 STRIDE_MAX 和最小的步进值 STRIDE_MIN 之差：

STRIDE_MAX　　STRIDE_MIN<=PASS_MAX

提问 1：如何证明该结论？有了该结论，在加上之前对优先级有 Priority > 1 限制，我们有

STRIDE_MAX　　STRIDE_MIN<=BIG_STRIDE

于是，只要将 BigStride 取在某个范围之内，即可保证对于任意两个 Stride 之差都会在机器整数表示的范围之内。可以通过其与 0 的比较结构，来得到两个 Stride 的大小关系。在上例中，虽然在直接的数值表示上 98<65535，但是 98−65535 的结果用带符号的 16 位整数表示的结果为 99，与理论值之差相等。所以在这个意义下 98>65535。基于这种特殊考虑的比较方法，即便 Stride 有可能溢出，我们仍能够得到理论上的当前最小 Stride，并做出正确的调度决定。

提问 2：在 ucore 中，目前 Stride 是采用无符号的 32 位整数表示，则 BigStride 应该取多少才能保证比较的正确性？

2. 使用优先队列实现 Stride Scheduling

在上述的实现描述中，对于每一次 pick_next 函数，都需要完整地扫描来获得当前最小的 stride 及其进程。这在进程非常多的时候是非常耗时和低效的，有兴趣的同学可以在实现了基于列表扫描的 Stride 调度器之后比较一下 priority 程序在 Round-Robin 及 Stride 调

度器下各自的运行时间。考虑到其调度选择与优先队列的抽象逻辑一致,我们考虑使用优化的优先队列数据结构实现该调度。

优先队列是这样一种数据结构:使用者可以快速地插入和删除队列中的元素,并且在预先指定的顺序下快速取得当前在队列中的最小(或者最大)值及其对应元素。可以看到,这样的数据结构非常符合 Stride 调度器的实现。

本次实验提供了 libs/skew_heap.h 作为优先队列的一个实现,该实现定义相关的结构和接口,其中主要包括如下:

```
//优先队列节点的结构
typedef struct skew_heap_entry  skew_heap_entry_t;
//初始化一个队列节点
void skew_heap_init(skew_heap_entry_t* a);
//将节点 b插入至以节点 a为队列头的队列中,返回插入后的队列
skew_heap_entry_t * skew_heap_insert(skew_heap_entry_t * a,
                                     skew_heap_entry_t * b,
                                     compare_f comp);
//将节点 b插入从以节点 a为队列头的队列中,返回删除后的队列
   skew_heap_entry_t* skew_heap_remove(skew_heap_entry_t* a,
                                       skew_heap_entry_t* b,
                                       compare_f comp);
```

其中优先队列的顺序是由比较函数 comp 决定的,sched_stride.c 中提供了 proc_stride_comp_f 比较器用来比较两个 stride 的大小,可以直接使用它。当使用优先队列作为 Stride 调度器的实现方式之后,运行队列结构也需要作相关改变,其中包括如下。

(1) struct run_queue 中的 lab6_run_pool 指针,在使用优先队列的实现中表示当前优先队列的头元素,如果优先队列为空,则其指向空指针(NULL)。

(2) struct proc_struct 中的 lab6_run_pool 结构,表示当前进程对应的优先队列节点。本次实验已经修改了系统相关部分的代码,使得其能够很好地适应 lab6 新加入的数据结构和接口。在实验中我们需要做的是用优先队列实现一个正确和高效的 Stride 调度器,如果用较简略的伪代码描述,则有如下表示。

① init(rq):

```
Initialize rq->run_list
Set rq->lab6_run_pool to NULL
Set rq->proc_num to 0
```

② enqueue(rq, proc):

```
Initialize proc->time_slice
Insert proc->lab6_run_pool into rq->lab6_run_pool
rq->proc_num++
```

③ dequeue(rq, proc):

```
Remove proc->lab6_run_pool from rq->lab6_run_pool
```

```
rq->proc_num--
```

④ pick_next(rq)：

```
If rq->lab6_run_pool==NULL, return NULL
Find the proc corresponding to the pointer rq->lab6_run_pool
proc->lab6_stride+=BIG_STRIDE / proc->lab6_priority
Return proc
```

⑤ proc_tick(rq, proc)：

```
If proc->time_slice>0, proc->time_slice--
If proc->time_slice==0, set the flag proc->need_resched
```

7.4 实验报告要求

从网站上下载 lab6.zip 后，解压得到本文档和代码目录 lab6，完成实验中的各个练习。完成代码编写并检查无误后，在对应目录下执行 make handin 任务，即会自动生成 lab6-handin.tar.gz。最后请一定提前或按时提交到网络学堂上。

注意有 lab6 的注释，主要是修改 default_sched_swide_c 中的内容。代码中所有需要完成的地方(challenge 除外)都有 lab6 和“Your Code”的注释，请在提交时特别注意保持注释，并将“Your Code”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

辅助材料 A 执行 priority 大致的显示输出

```
$make run-priority
…
check_swap() succeeded!
++setup timer interrupts
kernel_execve: pid=2, name="priority".
main: fork ok,now need to wait pids.
child pid 7, acc 2492000, time 2001
child pid 6, acc 1944000, time 2001
child pid 4, acc 960000, time 2002
child pid 5, acc 1488000, time 2003
child pid 3, acc 540000, time 2004
main: pid 3, acc 540000, time 2004
main: pid 4, acc 960000, time 2004
main: pid 5, acc 1488000, time 2004
main: pid 6, acc 1944000, time 2004
main: pid 7, acc 2492000, time 2004
main: wait pids over
stride sched correct result: 1 2 3 4 5
```

```
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:426:
    initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

第 8 章　实验 7：同步互斥

8.1　实 验 目 的

（1）熟悉 ucore 中的进程同步机制，了解操作系统为进程同步提供的底层支持。

（2）在 ucore 中理解信号量（semaphore）机制的具体实现。

（3）理解管程机制，在 ucore 内核中增加基于管程（monitor）的条件变量（condition variable）的支持。

（4）了解经典进程同步问题，并能使用同步机制解决进程同步问题。

8.2　实 验 内 容

实验 6 完成了用户进程的调度框架和具体的调度算法，可调度运行多个进程。如果多个进程需要协同操作或访问共享资源，则存在如何同步和有序竞争的问题。本次实验，主要是熟悉 ucore 的进程同步机制——信号量机制，以及基于信号量的哲学家就餐问题解决方案。然后掌握管程的概念和原理，并参考信号量机制，实现基于管程的条件变量机制和基于条件变量来解决哲学家就餐问题。

在本次实验中，在 kern/sync/check_sync.c 中提供了一个基于信号量的哲学家就餐问题解法。同时还需完成练习，即实现基于管程（主要是灵活运用条件变量和互斥信号量）的哲学家就餐问题解法。哲学家就餐问题描述如下：有五个哲学家，他们的生活方式是交替地进行思考和进餐。哲学家们共用一张圆桌，周围放有五把椅子，每人坐一把。在圆桌上有五个碗和五根筷子，当一个哲学家思考时，他不与其他人交谈，饥饿时便试图取用其左、右最靠近他的筷子，但他可能一根都拿不到。只有在他拿到两根筷子时，方能进餐，进餐完后，放下筷子又继续思考。

8.2.1　练习

练习 0：填写已有实验。

本实验依赖实验 1～实验 6。请把已做的实验 1～实验 6 的代码填入本实验的代码中有 lab1、lab2、lab3、lab4、lab5、lab6 的注释相应部分。并确保编译通过。

注意：为了能够正确执行 lab7 的测试应用程序，可能需对已完成的实验 1～实验 6 的代码进行进一步改进。

练习 1：理解内核级信号量的实现和基于内核级信号量的哲学家就餐问题（不需要编码）。

完成练习 0 后，建议大家比较一下（可用 kdiff3 等文件比较软件）个人完成的 lab6 和练习 0 完成后的刚修改的 lab7 之间的区别，分析了解 lab7 采用信号量的执行过程。执行 make grade，大部分测试用例应该通过。

练习 2：完成内核级条件变量和基于内核级条件变量的哲学家就餐问题（需要编码）。

首先掌握管程机制，然后基于信号量实现完成条件变量实现，然后用管程机制实现哲学家就餐问题的解决方案（基于条件变量）。

执行：make grade。如果所显示的应用程序检测都输出 ok，则基本正确。如果只是某程序过不去，比如 matrix. c，则可执行 make run-matrix 命令来单独调试它。大致执行结果可看附录 A（使用的是 qemu-1. 0. 1）。

扩展练习 Challenge：实现 Linux 的 RCU。

在 ucore 下实现下 Linux 的 RCU 同步互斥机制。可阅读相关 Linux 内核书籍或查询网上资料，可了解 RCU 的细节，然后大致实现在 ucore 中。下面是一些参考资料：

（1）http://www. ibm. com/developerworks/cn/linux/l-rcu/。

（2）http://www. diybl. com/course/6_system/linux/Linuxjs/20081117/151814. html。

8.2.2　项目组成

此次实验中，主要有如图 8-1 所示的一些需要关注的文件。

简单说明如下。

（1）kern/sync/sync. h：去除了 lock 实现（这对于不抢占内核没用）。

（2）kern/sync/wait. [ch]：定了为 wait 结构和 waitqueue 结构以及在此之上的函数，这是 ucore 中的信号量 semophore 机制和条件变量机制的基础，在本次实验中需要了解其实现。

（3）kern/sync/sem. [ch]:定义并实现了 ucore 中内核级信号量相关的数据结构和函数，本次实验中需要了解其中的实现，并基于此完成内核级条件变量的设计与实现。

（4）user/libs/{syscall. [ch]，ulib. [ch]}与 kern/sync/syscall. c：实现了进程 sleep 相关的系统调用的参数传递和调用关系。

（5）user/{sleep. c，sleepkill. c}：进程睡眠相关的一些测试用户程序。

（6）kern/sync/monitor. [ch]：基于管程的条件变量的实现程序，在本次实验中是练习的一部分，要求完成。

（7）kern/sync/check_sync. c：实现了基于管程的哲学家就餐问题，在本次实验中是练习的一部分，要求完成基于管程的哲学家就餐问题。

（8）kern/mm/vmm. [ch]：用信号量 mm_sem 取代 mm_struct 中原有的 mm_lock（本次实验不用管）。

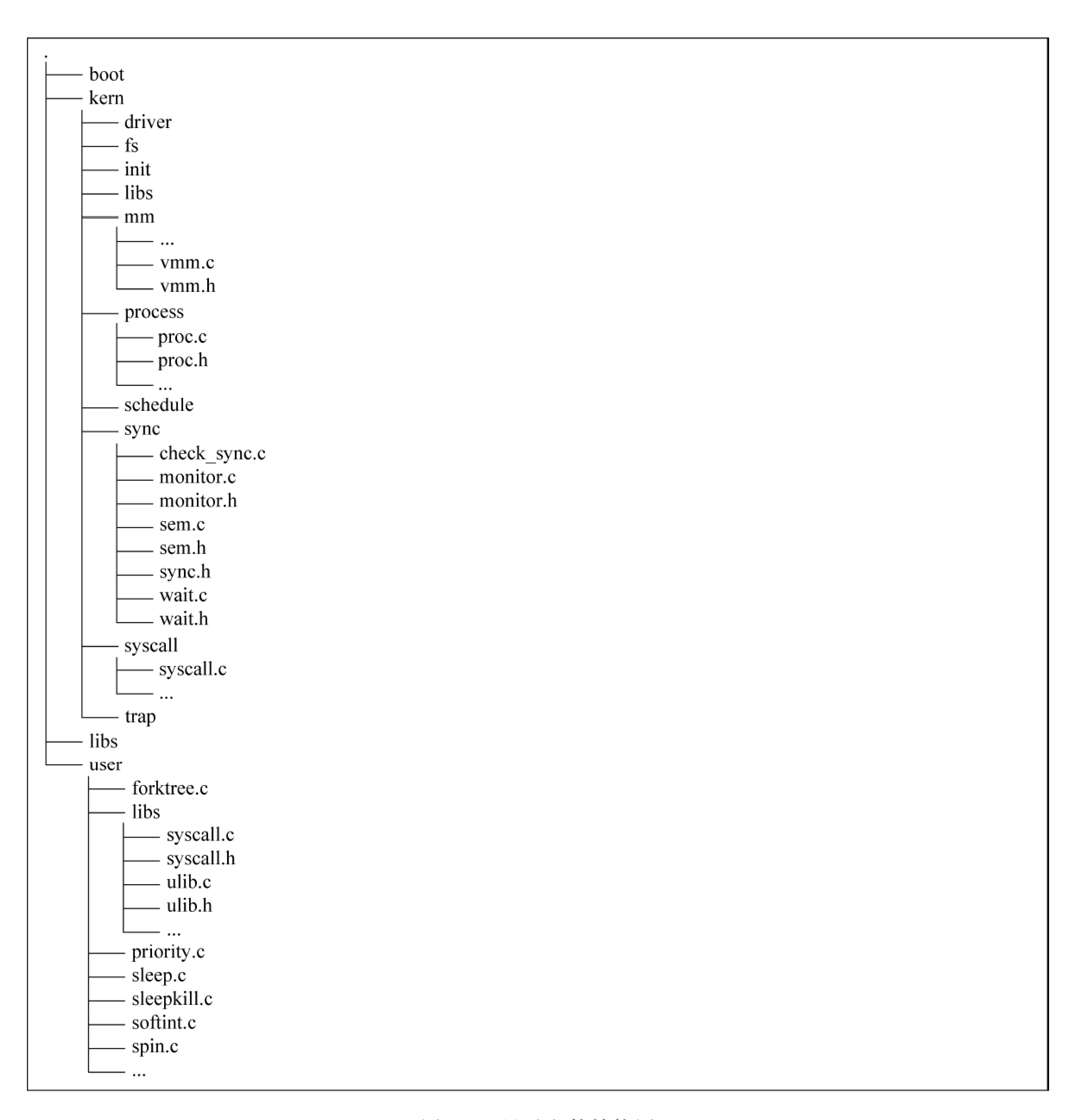


图 8-1　目录文件结构图

8.3　同步互斥的设计与实现

8.3.1　实验执行流程概述

互斥是指某一资源同时只允许一个进程对其进行访问，具有唯一性和排他性，但互斥不用限制进程对资源的访问顺序，即访问可以是无序的。同步是指在进程间的执行必须严格按照规定的某种先后次序来运行，即访问是有序的，这种先后次序取决于要系统完成的任务需求。在进程写资源情况下，进程间要求满足互斥条件。在进程读资源情况下，可允许多个进程同时访问资源。

实验 7 提供了多种同步互斥手段，包括中断控制、等待队列、信号量、管程机制（包含条件变量设计）等，并基于信号量实现了哲学家问题的执行过程，而练习是要求用管程机制实现哲学家问题的执行过程。在实现信号量机制和管程机制时，需要让无法进入临界区的进程睡眠，为此在 ucore 中设计了等待队列。当进程无法进入临界区（即无法获得信号量）时，可让进程进入等待队列，这时的进程处于等待状态（也可称为阻塞状态），从而会让实验 6 中的调度器选择一个处于就绪状态（即 RUNNABLE STATE）的进程，进行进程切换，让新进程有机会占用 CPU 执行，从而让整个系统的运行更加高效。

在实验 7 中的 ucore 初始化过程，开始的执行流程都与实验 6 相同，直到执行到创建第二个内核线程 init_main 时，修改了 init_main 的具体执行内容，即增加了 check_sync 函数的调用，而位于 lab7/kern/sync/check_sync.c 中的 check_sync 函数可以理解为是实验 7 的起始执行点，是实验 7 的总控函数。进一步分析此函数，可以看到这个函数主要分为两部分：第一部分是实现基于信号量的哲学家问题，第二部分是实现基于管程的哲学家问题。

对于 check_sync 函数的第一部分，首先实现初始化了一个互斥信号量，然后创建了对应 5 个哲学家行为的 5 个信号量，并创建 5 个内核线程代表 5 个哲学家，每个内核线程完成了基于信号量的哲学家吃饭、睡觉、思考行为实现。这部分是给学生作为练习参考用的。学生可以看看信号量是如何实现的，已经如何利用信号量完成哲学家问题。

对于 check_sync 函数的第二部分，首先初始化了管程，然后又创建了 5 个内核线程代表 5 个哲学家，每个内核线程要完成基于管程的哲学家吃饭、睡觉、思考行为实现。这部分需要学生来具体完成。学生需要掌握如何用信号量来实现条件变量，以及包含条件变量的管程如何确保哲学家能够正常思考和吃饭。

8.3.2 同步互斥的底层支撑

1. 开关中断

根据操作系统原理的知识，我们知道如果没有在硬件级保证读内存—修改值—写回内存的原子性，我们只能通过复杂的软件来实现同步互斥操作。但由于有开关中断和 test_and_set_bit 等原子操作机器指令的存在，使得我们在实现同步互斥原语上可以大大简化。在 atomic.c 文件中实现的 test_and_set_bit 等原子操作。

在 ucore 中提供的底层机制包括中断开关控制和 test_and_set 相关原子操作机器指令。kern/sync.c 中实现的开关中断的控制函数 local_intr_save(x)和 local_intr_restore(x)，它们是基于 kern/driver 文件下的 intr_enable()、intr_disable()函数实现的。具体调用关系为如下：

```
关中断：local_intr_save-->__intr_save-->intr_disable-->cli
开中断：local_intr_restore-->__intr_restore-->intr_enable-->sti
```

最终的 cli 和 sti 是 x86 的机器指令，它们实现了关中断和开中断，即设置了 eflags 寄存器中与中断相关的位。通过关闭中断，可以防止对当前执行的控制流被其他中断事件处理所打断。既然不能中断，那也就意味着在内核运行的当前进程无法被打断或被重新调度，即实现了对临界区的互斥操作。所以在单处理器情况下，可以通过开关中断实现对临界区的互斥保护，需要互斥的临界区代码的一般写法如下：

```
local_intr_save(intr_flag);
{
    临界区代码
}
local_intr_restore(intr_flag);
⋮
```

由于目前 ucore 只实现了对单处理器的支持，所以通过这种方式，就可简单地支撑互斥操作了。在多处理器情况下，这种方法是无法实现互斥的，因为屏蔽了一个 CPU 的中断，只能阻止本 CPU 上的进程不会被中断或调度，并不意味着其他 CPU 上执行的进程不能执行临界区的代码。所以，开关中断只对单处理器下的互斥操作起作用。在本实验中，开关中断机制是实现信号量等高层同步互斥原语的底层支撑基础之一。

2. 等待队列

到目前为止，我们的实验中，用户进程或内核线程还没有睡眠的支持机制。在课程中提到用户进程或内核线程可以转入休眠状态以等待某个特定事件，当该事件发生时这些进程能够被再次唤醒。内核实现这一功能的一个底层支撑机制就是等待队列（wait queue），等待队列和每一个事件（睡眠结束、时钟到达、任务完成、资源可用等）联系起来。需要等待事件的进程在转入休眠状态后插入到等待队列中。当事件发生之后，内核遍历相应等待队列，唤醒休眠的用户进程或内核线程，并设置其状态为就绪状态（runnable state），并将该进程从等待队列中清除。ucore 在 kern/sync/{wait. h，wait. c}中实现了 wait 结构和 wait queue 结构以及相关函数，这是实现 ucore 中的信号量机制和条件变量机制的基础，进入 wait queue 的进程会被设为睡眠状态，直到它们被唤醒。

```
typedef  struct {
    struct proc_struct * proc;          //等待进程的指针
    uint32_t wakeup_flags;              //进程被放入等待队列的原因标记
    wait_queue_t * wait_queue;          //指向此 wait 结构所属于的 wait_queue
    list_entry_t wait_link;             //用来组织 wait_queue 中 wait 节点的连接
} wait_t;
typedef struct {
    list_entry_t wait_head;             //wait_queue 的队头
} wait_queue_t;
le2wait(le, member)                     //实现 wait_t 中成员的指针向 wait_t 指针的转化
```

与 wait 和 wait queue 相关的函数主要分为两层，底层函数是对 wait queue 的初始化、插入、删除和查找操作，相关函数如下：

```
void wait_init(wait_t * wait, struct proc_struct * proc); //初始化 wait 结构
bool wait_in_queue(wait_t * wait);                        //wait 是否在等待队列 queue 中
void wait_queue_init(wait_queue_t * queue);               //初始化 wait_queue 结构
void wait_queue_add(wait_queue_t * queue, wait_t * wait); //把 wait 前插到 wait queue 中
void wait_queue_del(wait_queue_t * queue, wait_t * wait); //从 wait queue 中删除 wait
wait_t * wait_queue_next(wait_queue_t * queue, wait_t * wait);
                                                          //取得 wait 的后一个链接指针
```

```
wait_t* wait_queue_prev(wait_queue_t* queue, wait_t* wait);
                                                  //取得 wait 的前一个链接指针
wait_t* wait_queue_first(wait_queue_t* queue); //取得 wait queue 的第一个 wait
wait_t* wait_queue_last(wait_queue_t* queue);  //取得 wait queue 的最后一个 wait
bool wait_queue_empty(wait_queue_t* queue);    //wait queue 是否为空
```

高层函数基于底层函数实现了让进程进入等待队列,以及从等待队列中唤醒进程,相关函数如下:

```
//让 wait 与进程关联,且让当前进程关联的 wait 进入等待队列 queue,当前进程睡眠
void wait_current_set(wait_queue_t* queue, wait_t* wait, uint32_t wait_state);
//把与当前进程关联的 wait 从等待队列 queue 中删除
wait_current_del(queue, wait);
//唤醒与 wait 关联的进程
void wakeup_wait(wait_queue_t* queue, wait_t* wait, uint32_t wakeup_flags, bool del);
//唤醒等待队列上挂着的第一个 wait 所关联的进程
void wakeup_first(wait_queue_t* queue, uint32_t wakeup_flags, bool del);
//唤醒等待队列上所有的等待的进程
void wakeup_queue(wait_queue_t* queue, uint32_t wakeup_flags, bool del);
```

8.3.3 信号量

信号量是一种同步互斥机制的实现,普遍存在于现在的各种操作系统内核里。相对于 spinlock 的应用对象,信号量的应用对象是在临界区中运行的时间较长的进程。等待信号量的进程需要睡眠来减少占用 CPU 的开销。参考教科书"Operating Systems Internals and Design Principles"第 5 章"同步互斥"中对信号量实现的原理性描述:

```
struct semaphore {
    int count;
    queueType queue;
};
void semWait(semaphore s)
{
    s.count--;
    if (s.count< 0) {
    /* place this process in s.queue */;
    /* block this process */;
    }
}
void semSignal(semaphore s)
{
    s.count++;
    if (s.count<= 0) {
    /* remove a process P from s.queue */;
    /* place process P on ready list */;
    }
}
```

基于上述信号量实现可以认为,当多个(>1)进程可以进行互斥或同步合作时,一个进程会由于无法满足信号量设置的某条件而在某一位置停止,直到它接收到一个特定的信号(表明条件满足了)。为了发信号,需要使用一个称为信号量的特殊变量。为通过信号量 s 传送信号,信号量的 V 操作采用进程可执行原语 semSignal(s);为通过信号量 s 接收信号,信号量的 P 操作采用进程可执行原语 semWait(s);如果相应的信号仍然没有发送,则进程被阻塞或睡眠,直到发送完为止。

ucore 中信号量参照上述原理描述,建立在开关中断机制和 wait queue 的基础上进行了具体实现。信号量的数据结构定义如下:

```
typedef struct {
    int value;                        //信号量的当前值
    wait_queue_t wait_queue;          //信号量对应的等待队列
} semaphore_t;
```

semaphore_t 是最基本的记录型信号量(record semaphore)结构,包含了用于计数的整数值 value 和一个进程等待队列 wait_queue,一个等待的进程会挂在此等待队列上。

在 ucore 中最重要的信号量操作是 P 操作函数 down(semaphore_t * sem)和 V 操作函数 up(semaphore_t * sem)。但这两个函数的具体实现是__down(semaphore_t * sem, uint32_t wait_state) 函数和__up(semaphore_t * sem, uint32_t wait_state)函数,两者的具体实现描述如下。

(1) __down(semaphore_t * sem, uint32_t wait_state, timer_t * timer):具体实现信号量的 P 操作,首先关掉中断,然后判断当前信号量的 value 是否大于 0。如果是大于 0,则表明可以获得信号量,故让 value 减 1,并打开中断返回即可;如果不是大于 0,则表明无法获得信号量,故需要将当前的进程加入到等待队列中,并打开中断,然后运行调度器选择另外一个进程执行。如果被 V 操作唤醒,则把自身关联的 wait 从等待队列中删除(此过程需要先关中断,完成后开中断)。具体实现如下:

```
static __noinline uint32_t __down(semaphore_t * sem, uint32_t wait_state) {
    bool intr_flag;
    local_intr_save(intr_flag);
    if(sem->value>0) {
        sem->value--;
        local_intr_restore(intr_flag);
        return 0;
    }
    wait_t __wait, * wait=&__wait;
    wait_current_set(&(sem->wait_queue), wait, wait_state);
    local_intr_restore(intr_flag);

    schedule();

    local_intr_save(intr_flag);
    wait_current_del(&(sem->wait_queue), wait);
    local_intr_restore(intr_flag);
```

```
    if(wait->wakeup_flags != wait_state) {
        return wait->wakeup_flags;
    }
    return 0;
}
```

(2) __up(semaphore_t * sem, uint32_t wait_state)：具体实现信号量的 V 操作，首先关中断，如果信号量对应的 wait queue 中没有进程在等待，直接把信号量的 value 加 1，然后开中断返回；如果有进程在等待且进程等待的原因是 semophore 设置的，则调用 wakeup_wait 函数将 waitqueue 中等待的第一个 wait 删除，且把此 wait 关联的进程唤醒，最后开中断返回。具体实现如下：

```
static __noinline void __up(semaphore_t * sem, uint32_t wait_state) {
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        wait_t * wait;
        if ((wait=wait_queue_first(&(sem->wait_queue)))==NULL) {
            sem->value++;
        }
        else {
            wakeup_wait(&(sem->wait_queue), wait, wait_state, 1);
        }
    }
    local_intr_restore(intr_flag);
}
```

对照信号量的原理性描述和具体实现，可以发现两者在流程上基本一致，只是具体实现采用了关中断的方式保证了对共享资源的互斥访问，通过等待队列让无法获得信号量的进程睡眠等待。另外，我们可以看出信号量的计数器 value 具有如下性质。

① value>0，表示共享资源的空闲数。

② vlaue<0，表示该信号量的等待队列里的进程数。

③ value=0，表示等待队列为空。

8.3.4 管程和条件变量

引入了管程是为了将对共享资源的所有访问及其所需要的同步操作集中并封装起来。Hansan 为管程所下的定义：“一个管程定义了一个数据结构和能为并发进程所执行(在该数据结构上)的一组操作，这组操作能同步进程和改变管程中的数据”。有上述定义可知，管程由 4 部分组成。

(1) 管程内部的共享变量。

(2) 管程内部的条件变量。

(3) 管程内部并发执行的进程。

(4) 对局限在管程内部的共享数据设置初始值的语句。

局限在管程中的数据结构，只能被局限在管程的操作过程所访问，任何管程之外的操作过程都不能访问它；另一方面，局限在管程中的操作过程也主要访问管程内的数据结构。由此可见，管程相当于一个隔离区，它把共享变量和对它进行操作的若干个过程围了起来，所有进程要访问临界资源时，都必须经过管程才能进入，而管程每次只允许一个进程进入管程，从而需要确保进程之间互斥。

但在管程中仅仅有互斥操作是不够用的。进程可能需要等待某个条件 C 为真才能继续执行。如果采用忙等(busy waiting)方式：

```
while not ( C ) do{}
```

在单处理器情况下，将会导致所有其他进程都无法进入临界区使得该条件 C 为真，该管程的执行将会发生死锁。为此，可引入条件变量(Condition Variables，CV)。一个条件变量 CV 可理解为一个进程的等待队列，队列中的进程正等待某个条件 C 变为真。每个条件变量关联着一个断言 Pc。当一个进程等待一个条件变量，该进程不算作占用了该管程，因而其他进程可以进入该管程执行，改变管程的状态，通知条件变量 CV 其关联的断言 Pc 在当前状态下为真。因此对条件变量 CV 有两种主要操作。

① wait_cv：被一个进程调用，以等待断言 Pc 被满足后该进程可恢复执行. 进程挂在该条件变量上等待时，不被认为是占用了管程。

② signal_cv：被一个进程调用，以指出断言 Pc 现在为真，从而可以唤醒等待断言 Pc 被满足的进程继续执行。

有了互斥和信号量支持的管程就可用于解决各种同步互斥问题。比如参考 *OS Concept* 一书中的 6.7.2 节“用管程解决哲学家就餐问题”就给出了这样的事例：

```
monitor dp
{
    enum{THINKING,HUNGRY,EATING}state[5];
    condition self[5];

    void pickup(int i) {
        state[i]=HUNGRY;
        test(i);
        if(state[i]!=EATING)
            self[i].wait();
    }
    void putdown(int i) {
        state[i]=THINKING;
        test((i+4)%5);
        test((i+1%5));
    }

    void test(int i) {
        if(state[(i+4)%5]!=EATING)&&
        (state[i]==HUNGRY)&&
            (state[(i+1)%5]!=EATING){
```

```
            state[i]=EATING;
            self[i].signal();
        }
    }
    initialization_code() {
        for (int i=0;i<5; i++)
            state[i]=THINKING;
    }
}
```

虽然大部分教科书上说明管程适合在语言级实现,比如 Java 等高级语言,没有提及在采用 C 语言的 OS 中如何实现。下面我们将要尝试在 ucore 中用 C 语言实现采用基于互斥和条件变量机制的管程基本原理。

ucore 中的管程机制是基于信号量和条件变量来实现的。ucore 中的管程的数据结构 monitor_t 定义如下:

```
typedef struct monitor{
    semaphore_t mutex;      //the mutex lock for going into the routines in monitor, should be initialized
                              to 1
    semaphore_t next;       //the next semaphore is used to down the signaling proc itself, and the other
                              OR wakeuped
                            //waiting proc should wake up the sleeped signaling proc.
    int next_count;         //the number of sleeped signaling proc
    condvar_t * cv;         //the condvars in monitor
} monitor_t;
```

管程中的成员变量 mutex 是一个二值信号量,是实现每次只允许一个进程进入管程的关键元素,确保了互斥访问性质。管程中的条件变量 cv 通过执行 wait_cv,会使得等待某个条件 C 为真的进程能够离开管程并睡眠,且让其他进程进入管程继续执行;而进入管程的某进程设置条件 C 为真并执行 signal_cv 时,能够让等待某个条件 C 为真的睡眠进程被唤醒,从而继续进入管程中执行。管程中的成员变量信号量 next 和整型变量 next_count 是配合进程对条件变量 cv 的操作而设置的,这是由于发出 signal_cv 的进程 A 会唤醒睡眠进程 B,进程 B 执行会导致进程 A 睡眠,直到进程 B 离开管程,进程 A 才能继续执行,这个同步过程是通过信号量 next 完成的;而 next_count 表示了由于发出 singal_cv 而睡眠的进程个数。

管程中的条件变量的数据结构 condvar_t 定义如下:

```
typedef struct condvar{
    semaphore_t sem;        //the sem semaphore is used to down the waiting proc, and the signaling proc
                              should up the waiting proc
    int count;              //the number of waiters on condvar
    monitor_t * owner;      //the owner(monitor) of this condvar
} condvar_t;
```

条件变量的定义中也包含了一系列的成员变量,信号量 sem 用于让发出 wait_cv 操作的等待某个条件 C 为真的进程睡眠,而让发出 signal_cv 操作的进程通过这个 sem 来唤醒

睡眠的进程。count 表示等在这个条件变量上的睡眠进程的个数。owner 表示此条件变量的宿主是哪个管程。

理解数据结构的含义后，就可以开始管程的实现。ucore 设计实现了条件变量 wait_cv 操作和 signal_cv 操作对应的具体函数，即 cond_wait 函数和 cond_signal 函数，此外还有 cond_init 初始化函数(可直接看源码)。函数 cond_wait(condvar_t * cvp, semaphore_t * mp)和 cond_signal (condvar_t * cvp)的实现原理可参考 *OS Concept* 一书中的 6.7.3 节“用信号量实现管程”的内容。

cond_wait 的原理描述

```
cv.count++;
if(monitor.next_count>0)
     sem_signal(monitor.next);
else
     sem_signal(monitor.mutex);
sem_wait(cv.sem);
cv.count--;
```

Cond_signal 的原理描述

```
if(cv.count>0) {
     monitor.next_count++;
     sem_signal(cv.sem);
     sem_wait(monitor.next);
     monitor.next_count--;
}
```

简单分析一下 cond_wait 函数的实现。可以看出如果进程 A 执行了 cond_wait 函数，表示此进程等待某个条件 C 不为真，需要睡眠。因此表示等待此条件的睡眠进程个数 cv. count 要加 1。接下来会出现两种情况。

情况一：如果 monitor. next_count 大于 0，表示有大于等于 1 个进程执行 cond_signal 函数且睡着了，就睡在 monitor. next 信号量上。假定这些进程形成 S 进程链表。因此需要唤醒 S 进程链表中的一个进程 B。然后进程 A 睡在 cv. sem 上，如果睡醒了，则让 cv. count 减 1，表示等待此条件的睡眠进程个数少了一个，可继续执行了！这里隐含这一个现象，即某进程 A 在时间顺序上先执行了 signal_cv，而另一个进程 B 后执行了 wait_cv，这会导致进程 A 没有起到唤醒进程 B 的作用。这里还隐藏这一个问题，在 cond_wait 有 sem_signal(mutex)，但没有看到哪里有 sem_wait(mutex)，这好像没有成对出现，是否是错误的？其实在管程中的每一个函数的入口处会有 wait(mutex)，这样两者就配好对了。

情况二：如果 monitor. next_count 小于等于 0，表示目前没有进程执行 cond_signal 函数且睡着了，那需要唤醒的是由于互斥条件限制而无法进入管程的进程，所以要唤醒睡在 monitor. mutex 上的进程。然后进程 A 睡在 cv. sem 上，如果睡醒了，则让 cv. count 减 1，表示等待此条件的睡眠进程个数少了一个，可继续执行。

对照着再来看 cond_signal 的实现。首先进程 B 判断 cv. count，如果不大于 0，则表示当前没有执行 cond_wait 而睡眠的进程，因此就没有被唤醒的对象了，直接函数返回即可；如果大于 0，这表示当前有执行 cond_wait 而睡眠的进程 A，因此需要唤醒等待在 cv. sem 上睡眠的进程 A。由于只允许一个进程在管程中执行，所以一旦进程 B 唤醒了别人(进程 A)，那么自己就需要睡眠。故让 monitor. next_count 加 1，且让自己(进程 B)睡在信号量 monitor. next 上。如果睡醒了，这让 monitor. next_count 减 1。

为了让整个管程正常运行，还需在管程中的每个函数的入口和出口增加相关操作，即：

```
function (…)
```

```
{
sem.wait(monitor.mutex);

    the real body of function;

    if(monitor.next_count> 0)
        sem_signal(monitor.next);
    else
        sem_signal(monitor.mutex);
}
```

这样带来的作用有两个。

(1) 只有一个进程在执行管程中的函数。

(2) 避免由于执行了 cond_signal 函数而睡眠的进程无法被唤醒。对于第(2)点,如果进程 A 由于执行了 cond_signal 函数而睡眠(这会让 monitor. next_count 大于 0,且执行 sem_wait(monitor. next)),则其他进程在执行管程中的函数的出口,会判断 monitor. next_count 是否大于 0,如果大于 0,则执行 sem_signal(monitor. next),从而执行了 cond_signal 函数而睡眠的进程被唤醒。上述措施将使得管程正常执行。

需要注意的是,上述只是原理描述,与具体描述相比,还有一定的差距。需要大家在完成练习时仔细设计和实现。

8.4 实验报告要求

从网站上下载 lab7. zip 后,解压得到本文档和代码目录 lab7,完成实验中的各个练习。完成代码编写并检查无误后,在对应目录下执行 make handin 任务,即会自动生成 lab7-handin. tar. gz。最后请一定提前或按时提交到网络学堂上。

注意有 lab7 的注释,主要是修改 condvar. c 和 check_sync. c 中的内容。代码中所有需要完成的地方(Challenge 除外)都有 lab7 和"Your Code"的注释,请在提交时特别注意保持注释,并将"Your Code"替换为自己的学号,并且将所有标有对应注释的部分填上正确的代码。

辅助材料 A　执行 make run-matrix 大致的显示输出

```
(THU.CST) os is loading...
⋮
check_alloc_page() succeeded!
⋮
check_swap() succeeded!
++setup timer interrupts
I am No.4 philosopher_condvar
Iter 1, No.4 philosopher_condvar is thinking
I am No.3 philosopher_condvar
```

```
⋮
I am No.1 philosopher_sema
Iter 1, No.1 philosopher_sema is thinking
I am No.0 philosopher_sema
Iter 1, No.0 philosopher_sema is thinking
kernel_execve: pid=2, name="matrix".
pid 14 is running (1000 times)!.
pid 13 is running (1000 times)!.
phi_test_condvar: state_condvar[4] will eating
phi_test_condvar: signal self_cv[4]
Iter 1, No.4 philosopher_condvar is eating
phi_take_forks_condvar: 3 didn't get fork and will wait
phi_test_condvar: state_condvar[2] will eating
phi_test_condvar: signal self_cv[2]
Iter 1, No.2 philosopher_condvar is eating
phi_take_forks_condvar: 1 didn't get fork and will wait
phi_take_forks_condvar: 0 didn't get fork and will wait
pid 14 done!.
pid 13 done!.
Iter 1, No.4 philosopher_sema is eating
Iter 1, No.2 philosopher_sema is eating
⋮
pid 18 done!.
pid 23 done!.
pid 22 done!.
pid 33 done!.
pid 27 done!.
pid 25 done!.
pid 32 done!.
pid 29 done!.
pid 20 done!.
matrix pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:426:
    initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K> qemu: terminating on signal 2
```

第 9 章　实验 8：文件系统

9.1　实验目的

通过完成本次实验，希望能达到以下目标。

（1）了解基本的文件系统系统调用的实现方法。

（2）了解一个基于索引节点组织方式的 Simple FS 文件系统的设计与实现。

（3）了解文件系统抽象层——VFS 的设计与实现。

9.2　实验内容

实验 7 完成了在内核中的同步互斥实验。本次实验涉及的是文件系统，通过分析了解 ucore 文件系统的总体架构设计，完善读写文件操作，从新实现基于文件系统的执行程序机制（即改写 do_execve），从而可以完成执行存储在磁盘上的文件和实现文件读写等功能。

9.2.1　练习

练习 0：填写已有实验。

本实验依赖实验 1～实验 7。请把已做的实验 1～实验 7 的代码填入本实验中代码中有 lab1、lab2、lab3、lab4、lab5、lab6、lab7 的注释相应部分，并确保编译通过。注意：为了能够正确执行 lab8 的测试应用程序，可能需对已完成的实验 1～实验 7 的代码进行进一步改进。

练习 1：完成读文件操作的实现（需要编码）。

首先了解打开文件的处理流程，然后参考本实验后续的文件读写操作的过程分析，编写在 sfs_inode.c 中 sfs_io_nolock 读文件中数据的实现代码。

练习 2：完成基于文件系统的执行程序机制的实现（需要编码）。

改写 proc.c 中的 load_icode 函数和其他相关函数，实现基于文件系统的执行程序机制。执行：make qemu。如果能看看到 sh 用户程序的执行界面，则基本成功了。如果在 sh 用户界面上可以执行 ls，hello 等其他放置在 sfs 文件系统中的其他执行程序，则可以认为本实验基本成功（使用的是 qemu-1.0.1）。

9.2.2　项目组成

项目文件组成如图 9-1 所示。

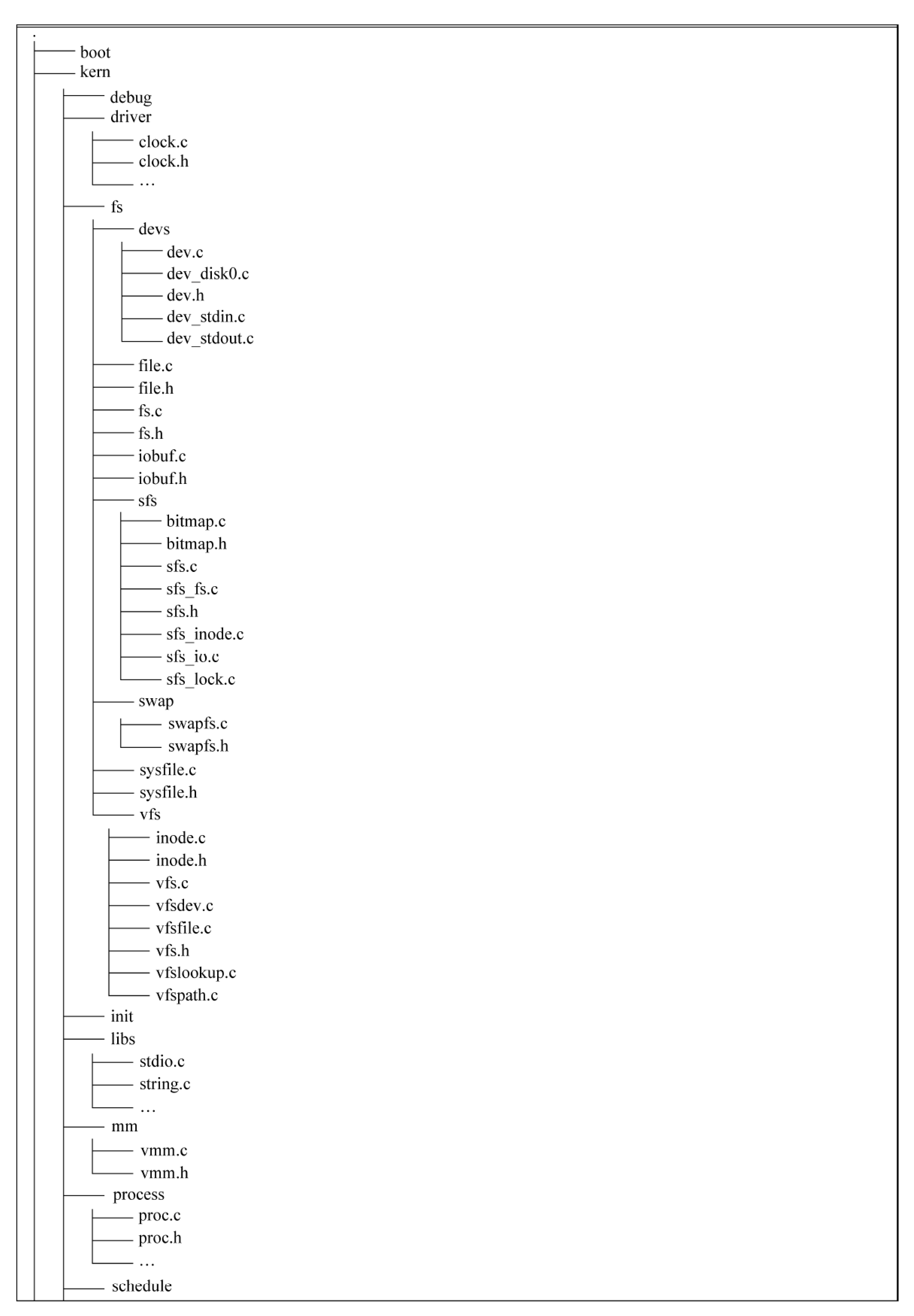

.
boot
kern
debug
driver
clock.c
clock.h
…
fs
devs
dev.c
dev_disk0.c
dev.h
dev_stdin.c
dev_stdout.c
file.c
file.h
fs.c
fs.h
iobuf.c
iobuf.h
sfs
bitmap.c
bitmap.h
sfs.c
sfs_fs.c
sfs.h
sfs_inode.c
sfs_io.c
sfs_lock.c
swap
swapfs.c
swapfs.h
sysfile.c
sysfile.h
vfs
inode.c
inode.h
vfs.c
vfsdev.c
vfsfile.c
vfs.h
vfslookup.c
vfspath.c
init
libs
stdio.c
string.c
…
mm
vmm.c
vmm.h
process
proc.c
proc.h
…
schedule

图 9-1　项目文件结构图

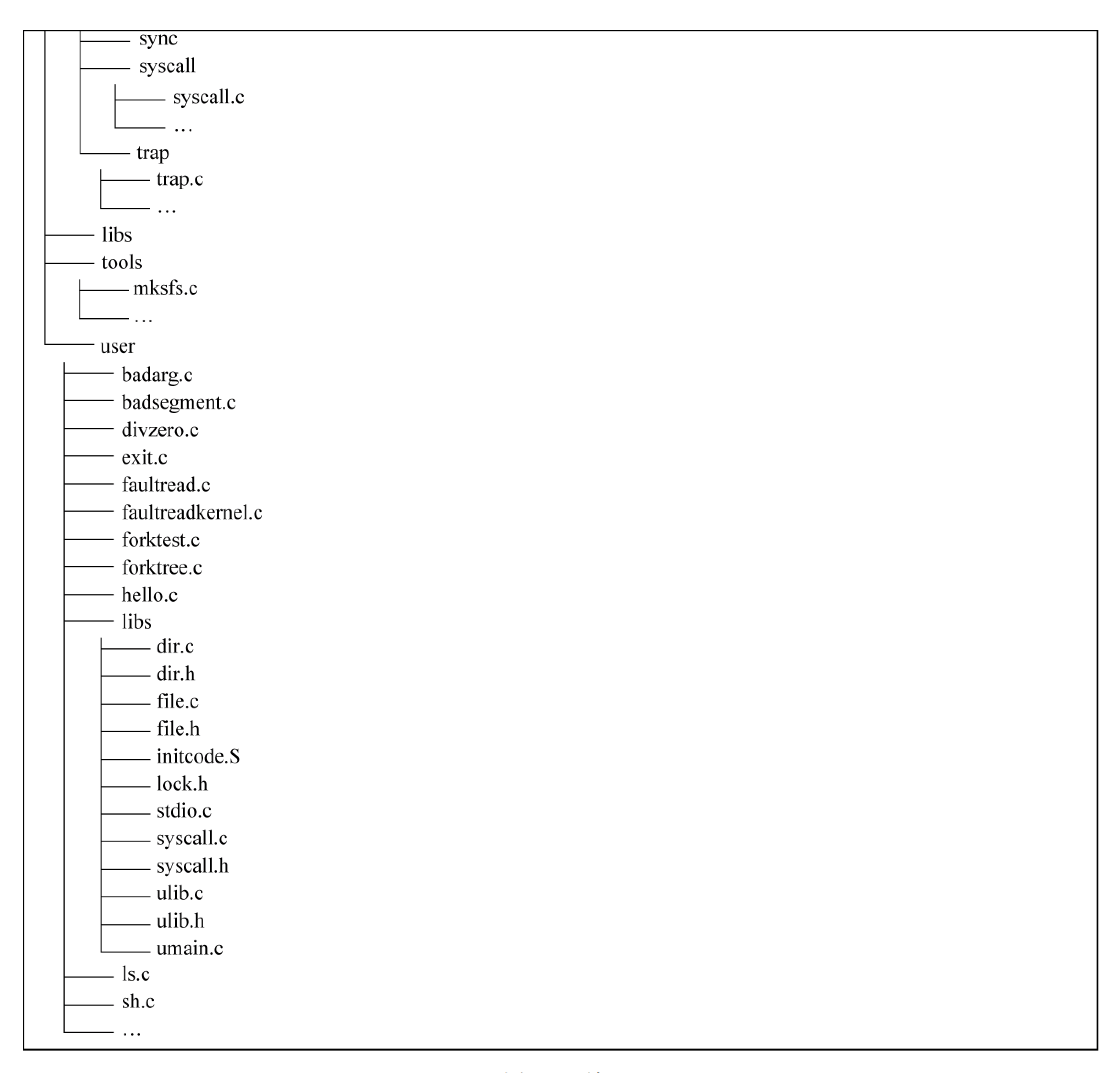


图 9-1（续）

本次实验主要是理解 kern/fs 目录中的部分文件，并可用 user/*.c 测试所实现的 Simple FS 文件系统是否能够正常工作。本次实验涉及的代码包括如下。

（1）文件系统测试用例——user/*.c：对文件系统的实现进行测试的测试用例。

（2）通用文件系统接口。

① user/libs/file.[ch]|dir.[ch]|syscall.c：与文件系统操作相关的用户库实现。

② kern/syscall.[ch]：文件中包含文件系统相关的内核态系统调用接口。

③ kern/fs/sysfile.[ch]|file.[ch]：通用文件系统接口和实现。

（3）文件系统抽象层——VFS。

kern/fs/vfs/*.[ch]：虚拟文件系统接口与实现。

（4）Simple FS 文件系统。

kern/fs/sfs/*.[ch]：SimpleFS 文件系统实现。

（5）文件系统的硬盘 I/O 接口。

kern/fs/devs/dev.[ch]|dev_disk0.c：disk0 硬盘设备提供给文件系统的 I/O 访问接口

和实现。

(6) 辅助工具。

tools/mksfs.c：创建一个 Simple FS 文件系统格式的硬盘镜像(理解此文件的实现细节对理解 SFS 文件系统很有帮助)。

(7) 对内核其他模块的扩充。

① kern/process/proc.[ch]：增加成员变量 struct fs_struct * fs_struct，用于支持进程对文件的访问；重写了 do_execve load_icode 等函数以支持执行文件系统中的文件。

② kern/init/init.c：增加调用初始化文件系统的函数 fs_init。

9.3 文件系统的设计与实现

9.3.1 ucore 文件系统总体介绍

操作系统中负责管理和存储可长期保存数据的软件功能模块称为文件系统。在本次实验中，主要侧重文件系统的设计实现和对文件系统执行流程的分析与理解。

ucore 的文件系统模型源于 Havard 的 OS161 的文件系统和 Linux 文件系统。但其实这两者都是源于传统的 UNIX 文件系统设计。UNIX 提出了 4 个文件系统抽象概念：文件(file)、目录项(dentry)、索引节点(inode)和安装点(mount point)。

(1) 文件：UNIX 文件中的内容可理解为是一有序字节 buffer，文件都有一个方便应用程序识别的文件名称(也称文件路径名)。典型的文件操作有读、写、创建和删除等。

(2) 目录项：目录项不是目录，而是目录的组成部分。在 UNIX 中目录被看做一种特定的文件，而目录项是文件路径中的一部分。如一个文件路径名是"/test/testfile"，则包含的目录项为根目录"/"、目录 test 和文件 testfile，这三个都是目录项。一般而言，目录项包含目录项的名字(文件名或目录名)和目录项的索引节点(见下面的描述)位置。

(3) 索引节点：UNIX 将文件的相关元数据信息(如访问控制权限、大小、拥有者、创建时间、数据内容等信息)存储在一个单独的数据结构中，该结构称为索引节点。

(4) 安装点：在 UNIX 中，文件系统被安装在一个特定的文件路径位置，这个位置就是安装点。所有的已安装文件系统都作为根文件系统树中的叶子出现在系统中。

上述抽象概念形成了 UNIX 文件系统的逻辑数据结构，并需要通过一个具体文件系统的架构设计与实现把上述信息映射并储存到磁盘介质上。一个具体的文件系统需要在磁盘布局并实现上述抽象概念。例如，文件元数据信息存储在磁盘块中的索引节点上。当文件被载入内存时，内核需要使用磁盘块中的索引点来构造内存中的索引节点。

ucore 模仿了 UNIX 的文件系统设计，ucore 的文件系统架构主要由四部分组成。

① 通用文件系统访问接口层：该层提供了一个从用户空间到文件系统的标准访问接口。这一层访问接口让应用程序能够通过一个简单的接口获得 ucore 内核的文件系统服务。

② 文件系统抽象层：向上提供一个一致的接口给内核其他部分(文件系统相关的系统调用实现模块和其他内核功能模块)访问。向下提供一个同样的抽象函数指针列表和数据结构屏蔽不同文件系统的实现细节。

③ Simple FS 文件系统层：一个基于索引方式的简单文件系统实例。向上通过各种具体函数实现以对应文件系统抽象层提出的抽象函数，向下访问外设接口。

④ 外设接口层：向上提供 device 访问接口屏蔽不同硬件细节。向下实现访问各种具

体设备驱动的接口，比如 disk 设备接口、串口设备接口、键盘设备接口等。

对照上面的层次，我们再大致介绍一下文件系统的访问处理过程，加深对文件系统的总体理解。假如应用程序操作文件（打开、创建、删除、读写），首先需要通过文件系统的通用文件系统访问接口层给用户空间提供的访问接口进入文件系统内部，接着由文件系统抽象层把访问请求转发给某一具体文件系统（如 SFS 文件系统），具体文件系统（Simple FS 文件系统层）把应用程序的访问请求转化为对磁盘上的 block 的处理请求，并通过外设接口层交给磁盘驱动例程来完成具体的磁盘操作。结合用户态写文件函数 write 的整个执行过程，可以比较清楚地看出 ucore 文件系统架构的层次和依赖关系，如图 9-2 所示。

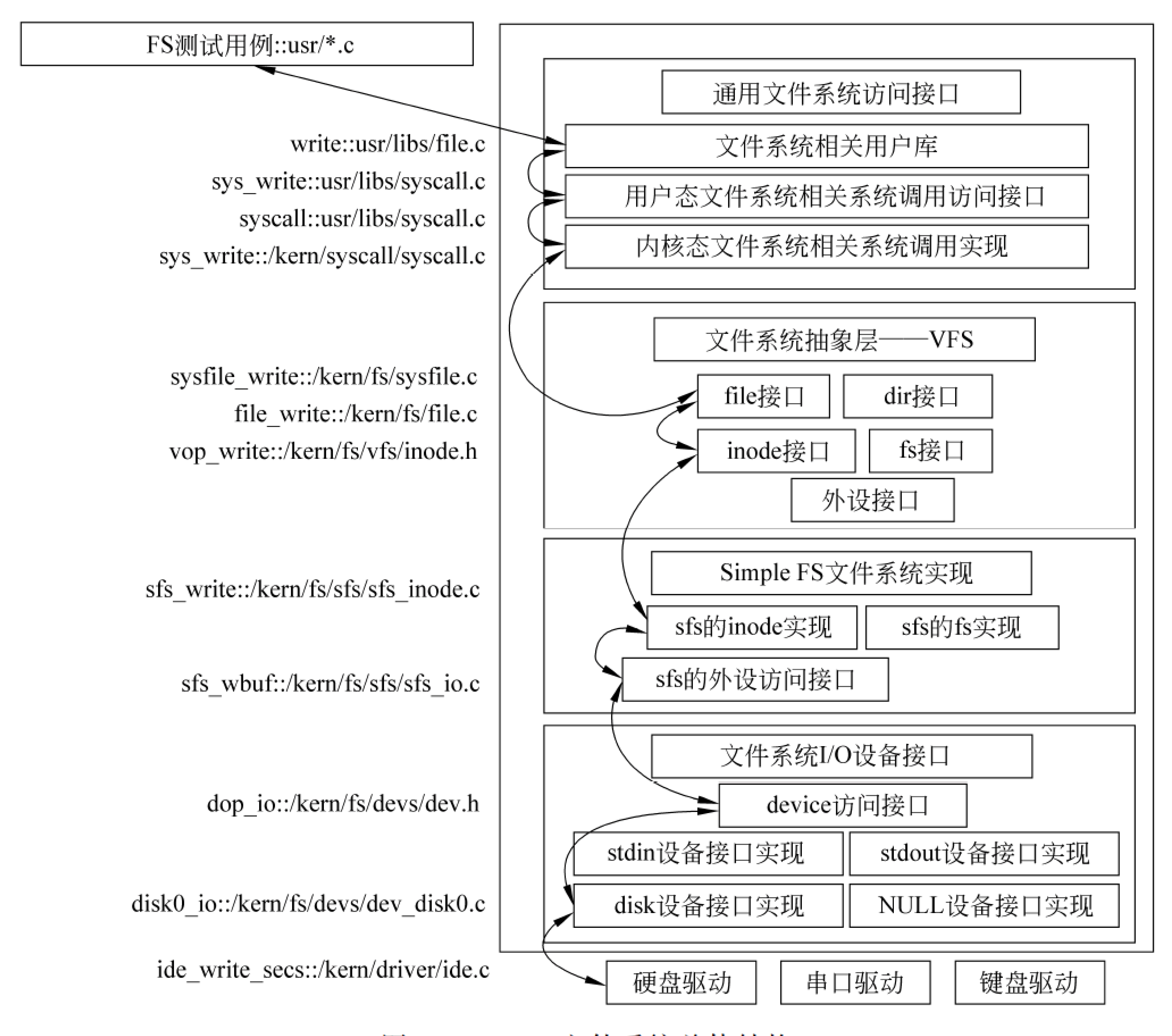


图 9-2　ucore 文件系统总体结构

从 ucore 操作系统不同的角度来看，ucore 中的文件系统架构包含四类主要的数据结构，它们分别如下。

（1）超级块（superblock）：它主要从文件系统的全局角度描述特定文件系统的全局信息。它的作用范围是整个 OS 空间。

（2）索引节点（inode）：它主要从文件系统的单个文件的角度描述文件的各种属性和数据所在位置。它的作用范围是整个 OS 空间。

（3）目录项（dentry）：它主要从文件系统的文件路径的角度描述文件路径中的特定目录。它的作用范围是整个 OS 空间。

（4）文件（file）：它主要从进程的角度描述一个进程在访问文件时需要了解的文件标识，文件读写的位置，文件引用情况等信息。它的作用范围是某一具体进程。

如果一个用户进程打开了一个文件，那么在 ucore 中涉及的相关数据结构(其中相关数据结构将在下面各个小节中展开叙述)和关系如图 9-3 所示。

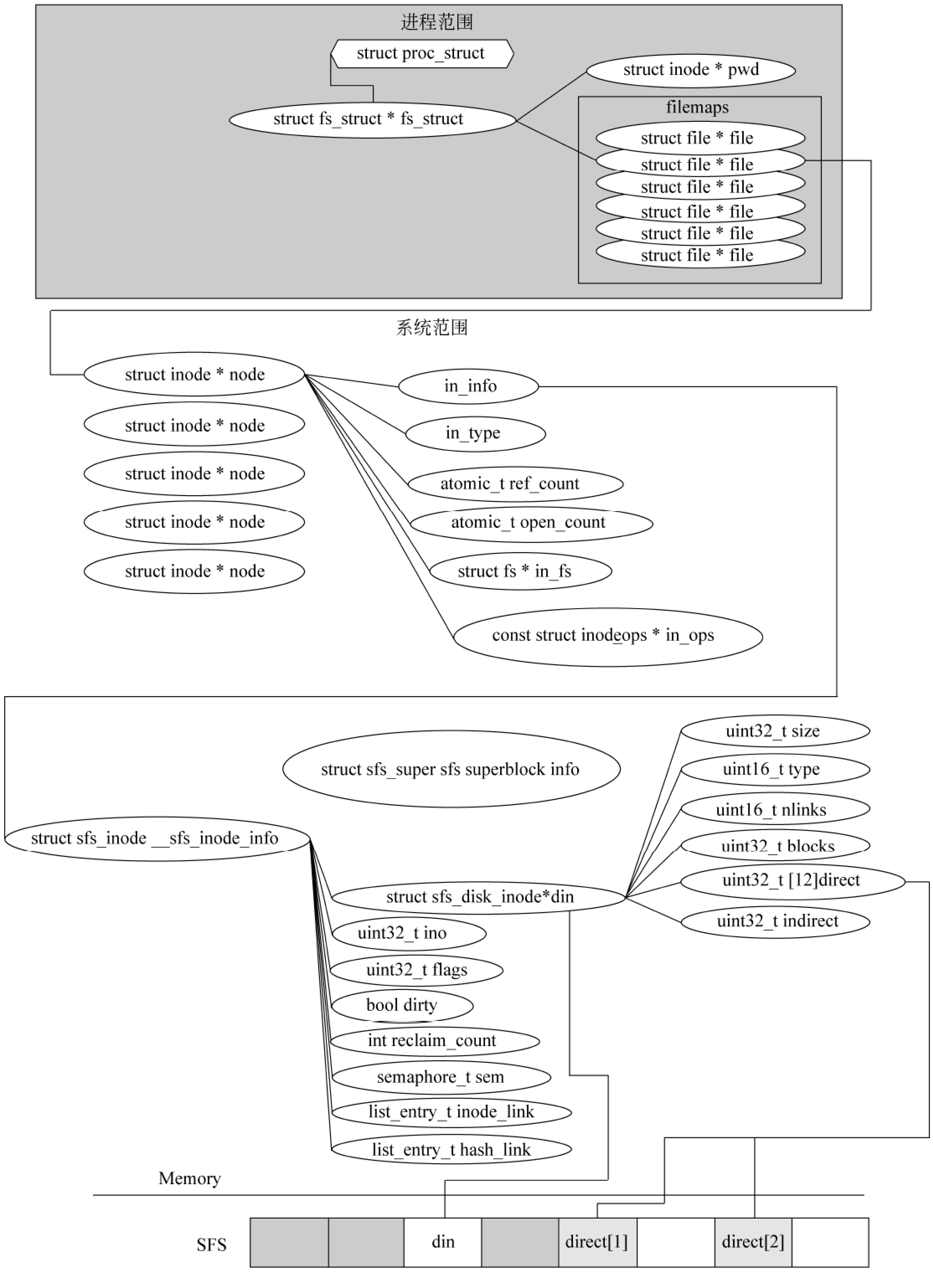


图 9-3　ucore 中文件相关关键数据结构及其关系

9.3.2 通用文件系统访问接口

1. 文件和目录相关用户库函数

lab8 中部分用户库函数与文件系统有关，我们先讨论对单个文件进行操作的系统调用，然后讨论对目录和文件系统进行操作的系统调用。

在文件操作方面，最基本的相关函数是 open、close、read、write。在读写一个文件之前，首先要用 open 系统调用将其打开。open 的第一个参数指定文件的路径名，可使用绝对路径名；第二个参数指定打开的方式，可设置为 O_RDONLY、O_WRONLY、O_RDWR，分别表示只读、只写、可读可写。打开一个文件后，就可以使用它返回的文件描述符 fd 对文件进行相关操作。使用完一个文件后，还要用 close 系统调用把它关闭，其参数就是文件描述符 fd。这样它的文件描述符就可以空出来，给别的文件使用。

读写文件内容的系统调用是 read 和 write。read 系统调用有三个参数：一个指定所操作的文件描述符，一个指定读取数据的存放地址，最后一个指定读多少个字节。在 C 程序中调用该系统调用的方法如下：

```
count= read(filehandle, buffer, nbytes);
```

该系统调用会把实际读到的字节数返回给 count 变量。在正常情形下这个值与 nbytes 相等，但有时可能会小一些。例如，在读文件时碰上了文件结束符，从而提前结束此次读操作。

如果由于参数无效或磁盘访问错误等原因，使得此次系统调用无法完成，则 count 被置为－1，而 write 函数的参数与之完全相同。

对于目录而言，最常用的操作是跳转到某个目录，这里对应的用户库函数是 chdir。然后就需要读目录的内容了，即列出目录中的文件或目录名，这在处理上与读文件类似，即需要通过 opendir 函数打开目录，通过 readdir 来获取目录中的文件信息，读完后还需通过 closedir 函数来关闭目录。由于在 ucore 中把目录看成一个特殊的文件，所以 opendir 和 closedir 实际上就是调用与文件相关的 open 和 close 函数。只有 readdir 需要调用获取目录内容的特殊系统调用 sys_getdirentry。而且这里没有写目录这一操作。在目录中增加内容其实就是在此目录中创建文件，需要用到创建文件的函数。

2. 文件和目录访问相关系统调用

与文件相关的 open、close、read、write 用户库函数对应的是 sys_open、sys_close、sys_read、sys_write 四个系统调用接口。与目录相关的 readdir 用户库函数对应的是 sys_getdirentry 系统调用。这些系统调用函数接口将通过 syscall 函数来获得 ucore 的内核服务。当到了 ucore 内核后，再调用文件系统抽象层的 file 接口和 dir 接口。

9.3.3 Simple FS 文件系统

这里我们没有按照从上到下先讲文件系统抽象层，再讲具体的文件系统。这是由于如果能够理解 Simple FS(SFS)文件系统，就可更好地分析文件系统抽象层的设计，即从具体走向抽象。ucore 内核把所有文件都看做字节流，任何内部逻辑结构都是专用的，由应用程序负责解释。但是 ucore 区分文件的物理结构。ucore 目前支持如下几种类型的文件。

(1) 常规文件：文件中包括的内容信息是由应用程序输入。SFS 文件系统在普通文件上不强加任何内部结构，把其文件内容信息看成字节。

(2) 目录：包含一系列的 entry，每个 entry 包含文件名和指向与之相关联的索引节点(index node)的指针。目录是按层次结构组织的。

(3) 链接文件：实际上一个链接文件是一个已经存在的文件的另一个可选择的文件名。

(4) 设备文件：不包含数据，但是提供了一个映射物理设备(如串口、键盘等)到一个文件名的机制。可通过设备文件访问外围设备。

(5) 管道：管道是进程间通信的一个基础设施。管道缓存了其输入端所接收的数据，以便在管道输出端读的进程能以一个先进先出的方式接收数据。

在 lab8 中关注的主要是 SFS 支持的常规文件、目录和链接中的 hardlink 的设计实现。SFS 文件系统中目录和常规文件具有共同的属性，而这些属性保存在索引节点中。SFS 通过索引节点来管理目录和常规文件，索引节点包含操作系统所需要的关于某个文件的关键信息，比如文件的属性、访问许可权以及其他控制信息都保存在索引节点中。可以有多个文件名指向一个索引节点。

1. 文件系统的布局

文件系统通常保存在磁盘上。在本实验中，第三个磁盘(即 disk0，前两个磁盘分别是 ucore.img 和 swap.img)用于存放一个 SFS 文件系统(Simple Filesystem)。文件系统中，磁盘的使用通常是以扇区(Sector)为单位的，但是为了实现简便，SFS 中以 block(4KB，与内存 page 大小相等)为基本单位。

SFS 文件系统的布局如下所示。

superblock	root-dir inode	freemap	inode/File Data/Dir Data blocks

第 0 个块(4KB)是超级块，它包含了关于文件系统的所有关键参数，当计算机启动或文件系统被首次接触时，超级块的内容就会被装入内存。其定义如下：

```
struct sfs_super {
    uint32_t magic;                     /* magic number, should be SFS_MAGIC */
    uint32_t blocks;                    /* # of blocks in fs */
    uint32_t unused_blocks;             /* # of unused blocks in fs */
    char info[SFS_MAX_INFO_LEN+1];      /* infomation for sfs  */
};
```

可以看到，它包含一个成员变量 magic，其值为 0x2f8dbe2a，内核通过它来检查磁盘镜像是否是合法的 SFS img；成员变量 blocks 记录了 SFS 中所有 block 的数量，即 img 的大小；成员变量 unused_block 记录了 SFS 中还没有被使用的 block 的数量；成员变量 info 包含了字符串"simple file system"。

第 1 个块放了一个 root-dir 的 inode，用来记录根目录的相关信息。有关 inode 还将在后续部分介绍。这里只要理解 root-dir 是 SFS 文件系统的根节点，通过这个 root-dir 的 inode 信息就可以定位并查找到根目录下的所有文件信息。

从第 2 个块开始，根据 SFS 中所有块的数量，用 1 个 bit 来表示一个块的占用和未被占

用的情况。这个区域称为 SFS 的 freemap 区域,这将占用若干个块空间。为了更好地记录和管理 freemap 区域,专门提供了两个文件 kern/fs/sfs/bitmap.[ch]来完成根据一个块号查找或设置对应的 bit 位的值。

最后在剩余的磁盘空间中,存放了所有其他目录和文件的 inode 信息和内容数据信息。需要注意的是,虽然 inode 的大小小于一个块的大小(4096B),但为了实现简单,每个 inode 都占用一个完整的 block。

在 sfs_fs.c 文件中的 sfs_do_mount 函数中,完成了加载位于硬盘上的 SFS 文件系统的超级块 super block 和 freemap 的工作。这样,在内存中就有了 SFS 文件系统的全局信息。

2. 索引节点

1) 磁盘索引节点

SFS 中的磁盘索引节点代表了一个实际位于磁盘上的文件。首先我们看看在硬盘上的索引节点的内容:

```
struct sfs_disk_inode {
    uint32_t size;                  /* 如果 inode 表示常规文件,则 size 是文件大小 */
    uint16_t type;                  /* inode 的文件类型 */
    uint16_t nlinks;                /* 此 inode 的硬链接数 */
    uint32_t blocks;                /* inode 的数据块数的个数 */
    uint32_t direct[SFS_NDIRECT];   /* 此 inode 的直接数据块索引值(有 SFS_NDIRECT 个) */
    uint32_t indirect;              /* 此 inode 的一级间接数据块索引值 */
};
```

从上面可以看出,如果 inode 表示的是文件,则成员变量 direct[]直接指向了保存文件内容数据的数据块索引值。indirect 间接指向了保存文件内容数据的数据块,indirect 指向的是间接数据块,此数据块实际存放的全部是数据块索引,这些数据块索引指向的数据块才被用来存放文件内容数据。

ucore 里 SFS_NDIRECT 默认是 12,即直接索引的数据页大小为 12×4KB=48KB;当使用一级间接数据块索引时,ucore 支持最大的文件大小为 12×4KB+1024×4KB=48KB+4MB。数据索引表内,0 表示一个无效的索引,inode 里 blocks 表示该文件或者目录占用的磁盘的 block 的个数。indiret 为 0 时,表示不使用一级索引块(因为 block 0 用来保存 super block,它不可能被其他任何文件或目录使用,所以这么设计也是合理的)。

对于普通文件,索引值指向的 block 中保存的是文件中的数据。对于目录,索引值指向的数据保存的是目录下所有的文件名以及对应的索引节点所在的索引块(磁盘块)所形成的数组。数据结构如下:

```
/* file entry (on disk) */
struct sfs_disk_entry {
    uint32_t ino;                            /* 索引节点所占数据块索引值 */
    char name[SFS_MAX_FNAME_LEN+1];          /* 文件名 */
};
```

操作系统中,每个文件系统下的 inode 都应该分配唯一的 inode 编号。在 SFS 下,为了实现的简便,每个 inode 直接用它所在的磁盘 block 的编号作为 inode 编号。例如,root

block 的 inode 编号为 1;每个 sfs_disk_entry 数据结构中,name 表示目录下文件或文件夹的名称,ino 表示磁盘 block 编号,通过读取该 block 的数据,能够得到相应的文件或文件夹的 inode。ino 为 0 时,表示一个无效的 entry。

此外,和 inode 相似,每个 sfs_dirent_entry 也占用一个 block。

2）内存中的索引节点

```
/* inode for sfs */
struct sfs_inode {
    struct sfs_disk_inode * din;          /* on-disk inode */
    uint32_t ino;                         /* inode number */
    uint32_t flags;                       /* inode flags */
    bool dirty;                           /* true if inode modified */
    int reclaim_count;                    /* kill inode if it hits zero */
    semaphore_t sem;                      /* semaphore for din */
    list_entry_t inode_link;              /* entry for linked-list in sfs_fs */
    list_entry_t hash_link;               /* entry for hash linked-list in sfs_fs */
};
```

可以看到,SFS 中的内存 inode 包含了 SFS 的硬盘 inode 信息,而且还增加了其他一些信息,这些信息便于进行判断是否改写、互斥操作、回收和快速地定位等作用。需要注意,一个内存 inode 是在打开一个文件后才创建的,如果关机则相关信息都会消失。而硬盘 inode 的内容是保存在硬盘中的,只是在进程需要时才被读入到内存中,用于访问文件或目录的具体内容数据。

为了方便实现上面提到的多级数据的访问以及目录中 entry 的操作,对 inode SFS 实现了一些辅助函数。

（1）sfs_bmap_load_nolock：将对应 sfs_inode 的第 index 个索引指向的 block 的索引值取出存到相应的指针指向的单元(ino_store)。该函数只接受 index＜＝inode－＞blocks 的参数。当 index＝＝inode－＞blocks 时,该函数理解为需要为 inode 增长一个 block,并标记 inode 为 dirty(所有对 inode 数据的修改都要做这样的操作,这样,当 inode 不再使用的时候,sfs 能够保证 inode 数据能够被写回到磁盘)。sfs_bmap_load_nolock 调用的 sfs_bmap_get_nolock 来完成相应的操作,阅读 sfs_bmap_get_nolock,了解它是如何工作的(sfs_bmap_get_nolock 只由 sfs_bmap_load_nolock 调用)。

（2）sfs_bmap_truncate_nolock：将多级数据索引表的最后一个 entry 释放掉。它可以认为是 sfs_bmap_load_nolock 中 index＝＝inode－＞blocks 的逆操作。当一个文件或目录被删除时,sfs 会循环调用该函数直到 inode－＞blocks 减为 0,释放所有的数据页。函数通过 sfs_bmap_free_nolock 来实现,它应该是 sfs_bmap_get_nolock 的逆操作。与 sfs_bmap_get_nolock 一样,调用 sfs_bmap_free_nolock 也要格外小心。

（3）sfs_dirent_read_nolock：将目录的第 slot 个 entry 读取到指定的内存空间。它通过上面提到的函数来完成。

（4）sfs_dirent_write_nolock：用指定的 entry 来替换某个目录下的第 slot 个 entry。它通过调用 sfs_bmap_load_nolock,保证当第 slot 个 entry 不存在时(slot＝＝inode－＞blocks),SFS 会分配一个新的 entry,即在目录尾添加了一个 entry。

(5) sfs_dirent_search_nolock：是常用的查找函数。它在目录下查找 name，并且返回相应的搜索结果(文件或文件夹)的 inode 的编号(也是磁盘编号)，以及相应的 entry 在该目录的 index 编号和目录下的数据页是否有空闲的 entry。SFS 中文件的数据页是连续的，不存在任何空洞；而对于目录，数据页不是连续的，当某个 entry 删除的时候，SFS 通过设置 entry－＞ino 为 0 将该 entry 所在的 block 标记为 free，在需要添加新 entry 的时候，SFS 优先使用这些 free 的 entry，其次才会去在数据页尾追加新的 entry。

注意：这些后缀为 nolock 的函数，只能在已经获得相应 inode 的 semaphore 时才能调用。

3) inode 的文件操作函数

```
static const struct inode_ops sfs_node_fileops= {
    .vop_magic              = VOP_MAGIC,
    .vop_open               = sfs_openfile,
    .vop_close              = sfs_close,
    .vop_read               = sfs_read,
    .vop_write              = sfs_write,
    ⋮
};
```

上述 sfs_openfile、sfs_close、sfs_read 和 sfs_write 分别对应用户进程发出的 open、close、read、write 操作。其中 sfs_openfile 不用做什么事；sfs_close 需要把对文件的修改内容写回到硬盘上，这样确保硬盘上的文件内容数据是最新的；sfs_read 和 sfs_write 函数都调用了一个函数 sfs_io，并最终通过访问硬盘驱动来完成对文件内容数据的读写。

4) inode 的目录操作函数

```
static const struct inode_ops sfs_node_dirops= {
    .vop_magic              = VOP_MAGIC,
    .vop_open               = sfs_opendir,
    .vop_close              = sfs_close,
    .vop_getdirentry        = sfs_getdirentry,
    .vop_lookup             = sfs_lookup,
    ⋮
};
```

对于目录操作而言，由于目录也是一种文件，所以 sfs_opendir、sys_close 对应户进程发出的 open、close 函数。相对于 sfs_open，sfs_opendir 只是完成一些 open 函数传递的参数判断，没做其他事情。目录的 close 操作与文件的 close 操作完全一致。由于目录的内容数据与文件的内容数据不同，所以读出目录的内容数据的函数是 sfs_getdirentry，其主要工作是获取目录下的文件 inode 信息。

9.3.4 文件系统抽象层——VFS

文件系统抽象层是把不同文件系统的对外共性接口提取出来，形成一个函数指针数组，这样，通用文件系统访问接口层只需访问文件系统抽象层，而不需关心具体文件系统的实现

细节和接口。

1. file 和 dir 接口

file 和 dir 接口层定义了进程在内核中直接访问的文件相关信息，这定义在 file 数据结构中，具体描述如下：

```
struct file {
    enum {
        FD_NONE, FD_INIT, FD_OPENED, FD_CLOSED,
    } status;                       //访问文件的执行状态
    bool readable;                  //文件是否可读
    bool writable;                  //文件是否可写
    int fd;                         //文件在 filemap 中的索引值
    off_t pos;                      //访问文件的当前位置
    struct inode * node;            //该文件对应的内存 inode 指针
    atomic_t open_count;            //打开此文件的次数
};
```

而在 kern/process/proc.h 中的 proc_struct 结构中描述了进程访问文件的数据接口 fs_struct，其数据结构定义如下：

```
struct fs_struct {
    struct inode * pwd;             //进程当前执行目录的内存 inode 指针
    struct file * filemap;          //进程打开文件的数组
    atomic_t fs_count;              //访问此文件的线程个数
    semaphore_t fs_sem;             //确保对进程控制块中 fs_struct 的互斥访问
};
```

当创建一个进程后，该进程的 fs_struct 将会被初始化或复制父进程的 fs_struct。当用户进程打开一个文件时，将从 filemap 数组中取得一个空闲 file 项，然后会把此 file 的成员变量 node 指针指向一个代表此文件的 inode 的起始地址。

2. inode 接口

index node 是位于内存的索引节点，它是 VFS 结构中的重要数据结构，因为它实际负责把不同文件系统的特定索引节点信息（甚至不能算是一个索引节点）统一封装起来，避免了进程直接访问具体文件系统。其定义如下：

```
struct inode {
    union {                                   //包含不同文件系统特定 inode 信息的 union 成员变量
        struct device __device_info;          //设备文件系统内存 inode 信息
        struct sfs_inode __sfs_inode_info;    //SFS 文件系统内存 inode 信息
    } in_info;
    enum {
        inode_type_device_info=0x1234,
        inode_type_sfs_inode_info,
    } in_type;                                //此 inode 所属文件系统类型
    atomic_t ref_count;                       //此 inode 的引用计数
    atomic_t open_count;                      //打开此 inode 对应文件的个数
```

```
    struct fs * in_fs;                        //抽象的文件系统,包含访问文件系统的函数指针
    const struct inode_ops * in_ops;          //抽象的 inode 操作,包含访问 inode 的函数指针
};
```

在 inode 中,有一成员变量为 in_ops,这是对此 inode 的操作函数指针列表,其数据结构定义如下:

```
struct inode_ops {
    unsigned long vop_magic;
    int(* vop_open)(struct inode * node, uint32_t open_flags);
    int(* vop_close)(struct inode * node);
    int(* vop_read)(struct inode * node, struct iobuf * iob);
    int(* vop_write)(struct inode * node, struct iobuf * iob);
    int(* vop_getdirentry)(struct inode * node, struct iobuf * iob);
    int(* vop_create)(struct inode * node, const char * name, bool excl, struct inode * * node_store);
    int(* vop_lookup)(struct inode * node, char * path, struct inode * * node_store);
    ⋮
};
```

参照上面对 SFS 中的索引节点操作函数的说明,可以看出 inode_ops 是对常规文件、目录、设备文件所有操作的一个抽象函数表示。对于某一具体的文件系统中的文件或目录,只需实现相关的函数,就可以被用户进程访问具体的文件,且用户进程无须了解具体文件系统的实现细节。

9.3.5 设备层文件 I/O 层

在本实验中,为了统一地访问设备,可以把一个设备看成一个文件,通过访问文件的接口来访问设备。目前实现了 stdin 设备文件、stdout 设备文件、disk0 设备。stdin 设备就是键盘,stdout 设备就是 CONSOLE(串口、并口和文本显示器),而 disk0 设备是承载 SFS 文件系统的磁盘设备。下面逐一分析 ucore 是如何让用户把设备看成文件来访问。

1. 关键数据结构

为了表示一个设备,需要有对应的数据结构,ucore 为此定义了 struct device,其描述如下:

```
struct device {
    size_t d_blocks;                                              //设备占用的数据块个数
    size_t d_blocksize;                                           //数据块的大小
    int (* d_open)(struct device * dev, uint32_t open_flags);     //打开设备的函数指针
    int (* d_close)(struct device * dev);                         //关闭设备的函数指针
    int (* d_io)(struct device * dev, struct iobuf * iob, bool write);
                                                                  //读写设备的函数指针
    int (* d_ioctl)(struct device * dev, int op, void * data);
                                                      //用 ioctl 方式控制设备的函数指针
};
```

这个数据结构能够支持对块设备(比如磁盘)、字符设备(比如键盘、串口)的表示,完成

对设备的基本操作。ucore 虚拟文件系统为了把这些设备链接在一起，还定义了一个设备链表，即双向链表 vdev_list，这样通过访问此链表，可以找到 ucore 能够访问的所有设备文件。

但这个设备描述没有与文件系统以及表示一个文件的 inode 数据结构建立关系，为此，还需要另外一个数据结构把 device 和 inode 联通起来，这就是 vfs_dev_t 数据结构：

```
//device info entry in vdev_list
typedef struct {
    const char* devname;
    struct inode* devnode;
    struct fs* fs;
    bool mountable;
    list_entry_t vdev_link;
} vfs_dev_t;
```

利用 vfs_dev_t 数据结构，就可以让文件系统通过一个链接 vfs_dev_t 结构的双向链表找到 device 对应的 inode 数据结构，一个 inode 节点的成员变量 in_type 的值是 0x1234，则此 inode 的成员变量 in_info 将成为一个 device 结构。这样 inode 就和一个设备建立了联系，这个 inode 就是一个设备文件。

2. stdout 设备文件

1）初始化

由于 stdout 设备是设备文件系统的文件，它自然有自己的 inode 结构。在系统初始化时，只需如下处理过程

```
kern_init-->fs_init-->dev_init-->dev_init_stdout-->dev_create_inode
                                                   -->stdout_device_init
                                                   -->vfs_add_dev
```

在 dev_init_stdout 中完成了对 stdout 设备文件的初始化。即首先创建了一个 inode，然后通过 stdout_device_init 完成对 inode 中的成员变量 inode—>__device_info 进行初始：

这里的 stdout 设备文件实际上就是指的 console 外设（它其实是串口、并口和 CGA 的组合型外设）。这个设备文件是一个只写设备，如果读这个设备，就会出错。接下来我们看看 stdout 设备的相关处理过程。

stdout 设备文件的初始化过程主要由 stdout_device_init 完成，其具体实现如下：

```
static void
stdout_device_init(struct device* dev) {
    dev->d_blocks=0;
    dev->d_blocksize=1;
    dev->d_open=stdout_open;
    dev->d_close=stdout_close;
    dev->d_io=stdout_io;
    dev->d_ioctl=stdout_ioctl;
}
```

可以看到，stdout_open 函数完成设备文件打开工作，如果发现用户进程调用 open 函数的参数 flags 不是只写(O_WRONLY)，则会报错。

2）访问操作实现

stdout_io 函数完成设备的写操作工作，具体实现如下：

```
static int
stdout_io(struct device * dev, struct iobuf * iob, bool write) {
    if (write) {
        char * data=iob->io_base;
        for (; iob->io_resid != 0; iob->io_resid--) {
            cputchar(* data++);
        }
        return 0;
    }
    return-E_INVAL;
}
```

可以看到，要写的数据放在 iob－>io_base 所指的内存区域，一直写到 iob－>io_resid 的值为 0 为止。每次写操作都是通过 cputchar 来完成的，此函数最终将通过 console 外设驱动来完成把数据输出到串口、并口和 CGA 显示器上。另外，也可以注意到，如果用户想执行读操作，则 stdout_io 函数直接返回错误值-E_INVAL。

3. stdin 设备文件

这里的 stdin 设备文件实际上就是指键盘。这个设备文件是一个只读设备，如果写这个设备，就会出错。接下来我们看看 stdin 设备的相关处理过程。

1）初始化

stdin 设备文件的初始化过程主要由 stdin_device_init 完成了主要的初始化工作，具体实现如下：

```
static void
stdin_device_init(struct device * dev) {
    dev->d_blocks=0;
    dev->d_blocksize=1;
    dev->d_open=stdin_open;
    dev->d_close=stdin_close;
    dev->d_io=stdin_io;
    dev->d_ioctl=stdin_ioctl;

    p_rpos=p_wpos=0;
    wait_queue_init(wait_queue);
}
```

相对于 stdout 的初始化过程，stdin 的初始化相对复杂一些，多了一个 stdin_buffer 缓冲区，描述缓冲区读写位置的变量 p_rpos、p_wpos 以及用于等待缓冲区的等待队列 wait_queue。在 stdin_device_init 函数的初始化中，也完成了对 p_rpos、p_wpos 和 wait_queue 的初始化。

2）访问操作实现

stdin_io 函数负责完成设备的读操作，具体实现如下：

```
static int
stdin_io(struct device * dev, struct iobuf * iob, bool write) {
    if (!write) {
        int ret;
        if ((ret=dev_stdin_read(iob->io_base, iob->io_resid))>0) {
            iob->io_resid-=ret;
        }
        return ret;
    }
    return-E_INVAL;
}
```

可以看到，如果是写操作，则 stdin_io 函数直接报错返回。所以这也进一步说明了此设备文件是只读文件。如果是读操作，则此函数进一步调用 dev_stdin_read 函数完成对键盘设备的读入操作。dev_stdin_read 函数的实现相对复杂一些，具体实现如下：

```
static int
dev_stdin_read(char * buf, size_t len) {
    int ret=0;
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        for (; ret<len; ret++, p_rpos++) {
        try_again:
            if (p_rpos<p_wpos) {
                * buf++=stdin_buffer[p_rpos % stdin_BUFSIZE];
            }
            else {
                wait_t __wait, * wait=&__wait;
                wait_current_set(wait_queue, wait, WT_KBD);
                local_intr_restore(intr_flag);

                schedule();

                local_intr_save(intr_flag);
                wait_current_del(wait_queue, wait);
                if (wait->wakeup_flags==WT_KBD) {
                    goto try_again;
                }
                break;
            }
        }
    }
    local_intr_restore(intr_flag);
```

```
    return ret;
}
```

在上述函数中可以看出，如果 p_rpos ＜ p_wpos，则表示有键盘输入的新字符在 stdin_buffer 中，于是就从 stdin_buffer 中取出新字符放到 iobuf 指向的缓冲区中；如果 p_rpos ＞＝p_wpos，则表明没有新字符，这样调用 read 用户态库函数的用户进程就需要采用等待队列的睡眠操作进入睡眠状态，等待从键盘输入字符。

从键盘输入字符后，如何唤醒等待键盘输入的用户进程呢？回顾 lab1 中的外设中断处理，可以了解到，当用户敲击键盘时，会产生键盘中断，在 trap_dispatch 函数中，当识别出中断是键盘中断（中断号为 IRQ_OFFSET＋IRQ_KBD）时，会调用 dev_stdin_write 函数，把字符写入 stdin_buffer 中，且会通过等待队列的唤醒操作唤醒正在等待键盘输入的用户进程。

9.3.6　实验执行流程概述

与实验 7 相比，实验 8 增加了文件系统，并因此实现了通过文件系统来加载可执行文件到内存中运行的功能，导致对进程管理相关的实现有比较大的调整。我们来看看文件系统是如何初始化并能在 ucore 的管理下正常工作。

首先看看 kern_init 函数，可以发现与 lab7 相比它增加了对 fs_init 函数的调用。fs_init 函数就是文件系统初始化的总控函数，它进一步调用了虚拟文件系统初始化函数 vfs_init，与文件相关的设备初始化函数 dev_init 和 Simple FS 文件系统的初始化函数 sfs_init。这三个初始化函数联合在一起，协同完成了整个虚拟文件系统、SFS 文件系统和文件系统对应的设备（键盘、串口、磁盘）的初始化工作。其函数调用关系图如图 9-4 所示。

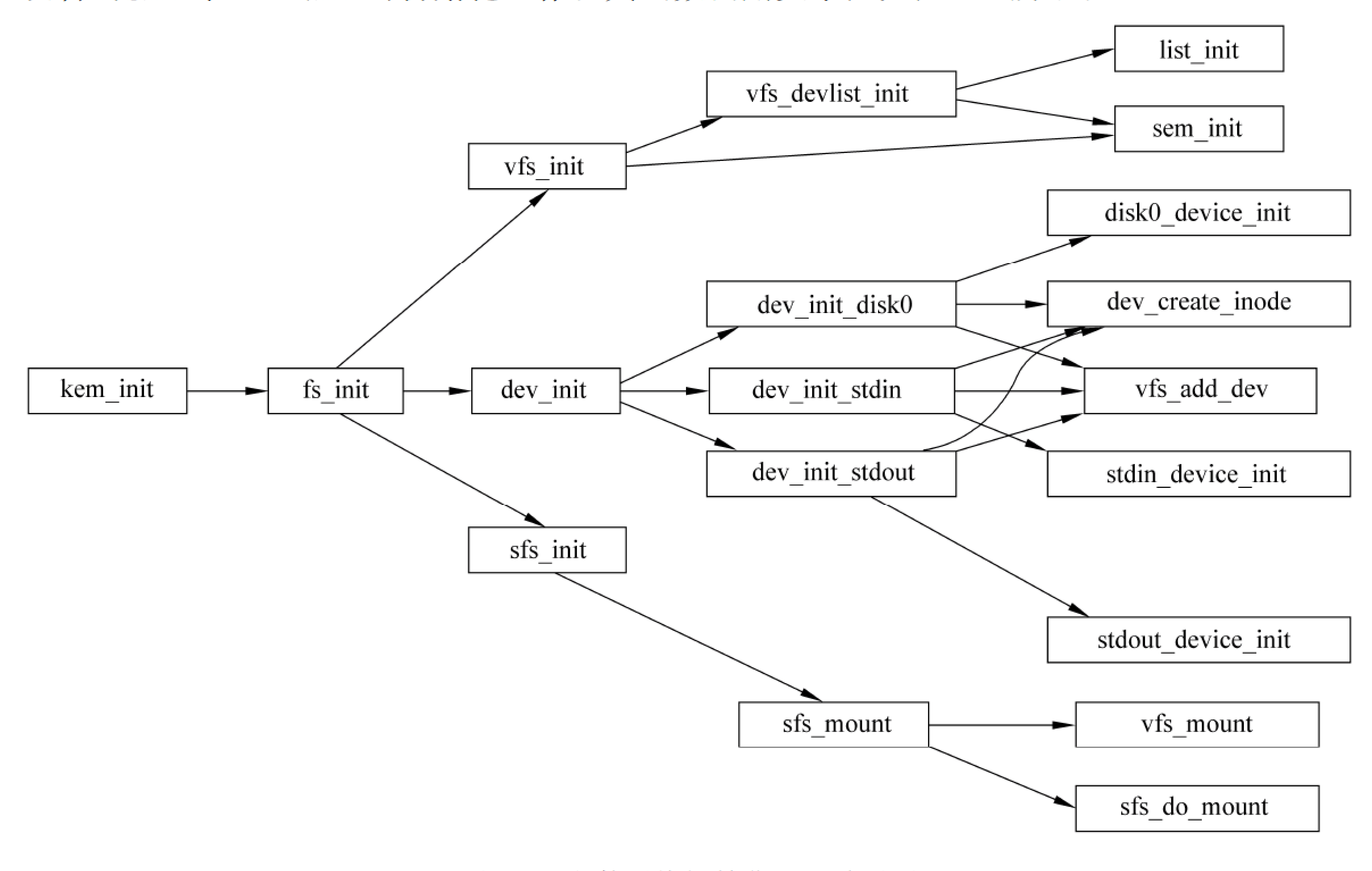


图 9-4　文件系统初始化调用关系图

参考图 9-4 并结合源码分析，可大致了解到文件系统的整个初始化流程。vfs_init 主要建立了一个 device list 双向链表 vdev_list，为后续具体设备（键盘、串口、磁盘）以文件的形式呈现建立查找访问通道。dev_init 函数通过进一步调用 disk0/stdin/stdout_device_init 完成对具体设备的初始化，把它们抽象成一个设备文件，并建立对应的 inode 数据结构，最后把它们链入 vdev_list 中。这样通过虚拟文件系统就可以方便地以文件的形式访问这些设备了。sfs_init 是完成对 Simple FS 的初始化工作，并把此实例文件系统挂在虚拟文件系统中，从而让 ucore 的其他部分能够通过访问虚拟文件系统的接口来进一步访问 SFS 实例文件系统。

9.3.7 文件操作实现

1. 打开文件

有了上述分析后，我们可以看看如果一个用户进程打开文件会做哪些事情？首先假定用户进程需要打开的文件已经存在硬盘上。以 user/sfs_filetest1. c 为例，首先用户进程会调用在 main 函数中的如下语句：

```
int fd1=safe_open("/test/testfile",O_RDWR|O_TRUNC);
```

从字面上可以看出，如果 ucore 能够正常查找到这个文件，就会返回一个代表文件的文件描述符 fd1，这样在接下来的读写文件过程中，就直接用 fd1 来代表就可以。那么这个打开文件的过程是如何一步一步实现的呢？

1）通用文件访问接口层的处理流程

首先进入通用文件访问接口层的处理流程，即进一步调用如下用户态函数：open－＞sys_open－＞syscall，从而引起系统调用进入到内核态。到了内核态后，通过中断处理例程，会调用到 sys_open 内核函数，并进一步调用 sysfile_open 内核函数。到了这里，需要把位于用户空间的字符串“/test/testfile”复制到内核空间中的字符串 path 中，并进入文件系统抽象层的处理流程完成进一步的打开文件操作。

2）文件系统抽象层的处理流程

（1）分配一个空闲的 file 数据结构变量 file。

在文件系统抽象层的处理中，首先调用的是 file_open 函数，它要给这个即将打开的文件分配一个 file 数据结构的变量，这个变量其实是当前进程的打开文件数组 current－＞fs_struct－＞filemap[]中的一个空闲元素（即还没用于一个打开的文件），而这个元素的索引值就是最终要返回到用户进程并赋值给变量 fd1。到了这一步还仅仅是给当前用户进程分配了一个 file 数据结构的变量，还没有找到对应的文件索引节点。

为此需要进一步调用 vfs_open 函数来找到 path 指出的文件所对应的基于 inode 数据结构的 VFS 索引节点 node。vfs_open 函数需要完成两件事情：通过 vfs_lookup 找到 path 对应文件的 inode；调用 vop_open 函数打开文件。

（2）找到文件设备的根目录“/”的索引节点。

需要注意，这里的 vfs_lookup 函数是一个针对目录的操作函数，它会调用 vop_lookup 函数来找到 SFS 文件系统中的“/test”目录下的 testfile 文件。为此，vfs_lookup 函数首先调用 get_device 函数，并进一步调用 vfs_get_bootfs 函数（其实调用了）来找到根目录“/”对

应的 inode。这个 inode 就是位于 vfs.c 中的 inode 变量 bootfs_node。这个变量在 init_main 函数(位于 kern/process/proc.c)执行时获得了赋值。

(3) 找到根目录"/"下的 test 子目录对应的索引节点。

在找到根目录对应的 inode 后,通过调用 vop_lookup 函数来查找"/"和 test 这两层目录下的文件 testfile 所对应的索引节点,如果找到就返回此索引节点。

(4) 把 file 和 node 建立联系。

完成第(3)步后,将返回到 file_open 函数中,通过执行语句"file－＞node＝node;",就把当前进程的 current－＞fs_struct－＞filemap[fd](即 file 所指变量)的成员变量 node 指针指向了代表"/test/testfile"文件的索引节点 node。这时返回 fd。经过重重回退,通过系统调用返回,用户态的 syscall－＞sys_open－＞open－＞safe_open 等用户函数的层层函数返回,最终把把 fd 赋值给 fd1。自此完成了打开文件操作。但这里还没有分析第(2)步和第(3)步是如何进一步调用 SFS 文件系统提供的函数,并利用该函数找位于 SFS 文件系统上的"/test/testfile"所对应的 sfs 磁盘 inode 的过程。下面需要进一步对此进行分析。

3) SFS 文件系统层的处理流程

这里需要分析文件系统抽象层中没有彻底分析的 vop_lookup 函数到底做了什么。下面我们来看看。在 sfs_inode.c 中的 sfs_node_dirops 变量定义了.vop_lookup ＝ sfs_lookup,所以我们重点分析 sfs_lookup 的实现。

sfs_lookup 有三个参数:node、path、node_store。其中 node 是根目录"/"所对应的 inode 节点;path 是文件 testfile 的绝对路径/test/testfile,而 node_store 是经过查找获得的 testfile 所对应的 inode 节点。

Sfs_lookup 函数以"/"为分割符,从左至右逐一分解 path 获得各个子目录和最终文件对应的 inode 节点。在本例中是分解出 test 子目录,并调用 sfs_lookup_once 函数获得 test 子目录对应的 inode 节点 subnode,然后循环进一步调用 sfs_lookup_once 查找以 test 子目录下的文件 testfile1 所对应的 inode 节点。当无法分解 path 后,就意味着找到了 testfile1 对应的 inode 节点,就可顺利返回了。

当然这里讲得还比较简单,sfs_lookup_once 将调用 sfs_dirent_search_nolock 函数来查找与路径名匹配的目录项,如果找到目录项,则根据目录项中记录的 inode 所处的数据块索引值找到路径名对应的 SFS 磁盘 inode,并读入 SFS 磁盘 inode 对的内容,创建 SFS 内存 inode。

2. 读文件

读文件其实就是读出目录中的目录项,首先假定文件在磁盘上且已经打开。用户进程有如下语句:

```
read(fd, data, len);
```

即读取 fd 对应文件,读取长度为 len,存入 data 中。下面来分析一下读文件的实现。

1) 通用文件访问接口层的处理流程

先进入通用文件访问接口层的处理流程,即进一步调用如下用户态函数:read－＞sys_read－＞syscall,从而引起系统调用进入到内核态。到了内核态以后,通过中断处理例程,会调用到 sys_read 内核函数,并进一步调用 sysfile_read 内核函数,进入文件系统抽象层处

理流程完成进一步读文件的操作。

2）文件系统抽象层的处理流程

（1）检查错误，即检查读取长度是否为 0 和文件是否可读。

（2）分配 buffer 空间，即调用 kmalloc 函数分配 4096B 的 buffer 空间。

（3）读文件过程。

① 实际读文件。

循环读取文件，每次读取 buffer 大小。每次循环中，先检查剩余部分大小，若其小于 4096B，则只读取剩余部分的大小。然后调用 file_read 函数（详细分析见后面的内容）将文件内容读取到 buffer 中，alen 为实际大小。调用 copy_to_user 函数将读到的内容复制到用户的内存空间中，调整各变量以进行下一次循环读取，直至指定长度读取完成。最后函数调用层层返回至用户程序，用户程序收到了读到的文件内容。

② file_read 函数。

这个函数是读文件的核心函数。它有 4 个参数，fd 是文件描述符，base 是缓存的基地址，len 是要读取的长度，copied_store 存放实际读取的长度。函数首先调用 fd2file 函数找到对应的 file 结构，并检查是否可读。调用 filemap_acquire 函数使打开这个文件的计数加 1。调用 vop_read 函数将文件内容读到 iob 中（详细分析见后）。调整文件指针偏移量 pos 的值，使其向后移动实际读到的字节数 iobuf_used(iob)。最后调用 filemap_release 函数使打开这个文件的计数减 1，若打开计数为 0，则释放 file。

3）SFS 文件系统层的处理流程

vop_read 函数实际上是对 sfs_read 的包装。在 sfs_inode.c 中 sfs_node_fileops 变量定义了.vop_read = sfs_read，所以下面来分析 sfs_read 函数的实现。

sfs_read 函数调用 sfs_io 函数。它有三个参数，node 是对应文件的 inode，iob 是缓存，write 表示是读还是写的布尔值（0 表示读，1 表示写），这里是 0。函数先找到 inode 对应 sfs 和 sin，然后调用 sfs_io_nolock 函数进行读取文件操作，最后调用 iobuf_skip 函数调整 iobuf 的指针。

在 sfs_io_nolock 函数中，先计算一些辅助变量，并处理一些特殊情况（比如越界），然后有 sfs_buf_op = sfs_rbuf，sfs_block_op = sfs_rblock，设置读取的函数操作。接着进行实际操作，先处理起始的没有对齐到块的部分，再以块为单位循环处理中间的部分，最后处理末尾剩余的部分。每部分中都调用 sfs_bmap_load_nolock 函数得到 blkno 对应的 inode 编号，并调用 sfs_rbuf 或 sfs_rblock 函数读取数据（中间部分调用 sfs_rblock，起始和末尾部分调用 sfs_rbuf），调整相关变量。完成后如果 offset＋alen ＞ din－＞fileinfo.size（写文件时会出现这种情况，读文件时不会出现这种情况，alen 为实际读写的长度），则调整文件大小为 offset＋alen 并设置 dirty 变量。

sfs_bmap_load_nolock 函数将对应 sfs_inode 的第 index 个索引指向的 block 的索引值取出存到相应的指针指向的单元（ino_store）。它调用 sfs_bmap_get_nolock 来完成相应的操作。sfs_rbuf 和 sfs_rblock 函数最终都调用 sfs_rwblock_nolock 函数完成操作，而 sfs_rwblock_nolock 函数调用 dop_io－＞disk0_io－＞disk0_read_blks_nolock－＞ide_read_secs 完成对磁盘的操作。

9.4 实验报告要求

从网站上下载 lab8.zip 后，解压得到本文档和代码目录 lab8，完成实验中的各个练习。完成代码编写并检查无误后，在对应目录下执行 make handin 任务，即会自动生成 lab8-handin.tar.gz。最后请一定提前或按时提交到网络学堂上。

注意有 lab8 的注释，这是需要主要修改的内容。代码中所有需要完成的地方 Challenge 除外）都有 lab8 和"Your Code"的注释，请在提交时特别注意保持注释，并将"Your Code"替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。